MEMORIA DEL PROYECTO

FIN DE CARRERA

Arquitectura software y hardware para la automatización de una carretilla industrial

Departamento de Informática y Automática

Facultad de Ciencias

Universidad de Salamanca

Titulación: Ingeniería en Informática (2º ciclo)

Autora: Raquel Sánchez Díaz

Tutores: Belén Curto Diego y Francisco Javier Blanco Rodríguez

Fecha: Junio 2007

Tabla de Contenidos

1. INTRODUCCIÓN 1

1.1. BREVE DESCRIPCIÓN DE CONTENIDOS 4
1.2. ESTRUCTURA DEL CD-ROM 5

2. OBJETIVOS 7

2.1. OBJETIVOS DEL PROYECTO 7
2.2. OBJETIVOS PERSONALES 8

3. CONCEPTOS TEÓRICOS 9

3.1. MICROCONTROLADORES 9
3.1.1 PIC18F258 10
3.2. PLACA SBC28PC 11
3.3. SENSOR DE ULTRASONIDO SRF08 12
3.3.1 Funcionamiento 13
3.3.2 Direcciones 14
3.3.3 Conexiones 14
3.3.4 Registros 15
3.3.5 Instrucciones 16
3.3.6 Mediciones 16
3.4. BUS 17
3.5. RS485 17
3.5.1 VScom USB-COM-I 17
3.6. BUS I2C 17
3.6.1 Características principales 18
3.6.2 Protocolo de comunicaciones 18
3.6.3 Direccionamiento en el bus I2C 20
3.6.4 Escritura en un dispositivo esclavo 20
3.6.5 Lectura desde un dispositivo esclavo 21
3.7. BUS DE CAMPO 22
3.8. BUS CAN 23
3.8.1 Componentes del sistema Can-Bus 24
3.8.2 Principales características del protocolo CAN 26
3.8.3 Funcionamiento del Can-Bus 27
3.8.4 Estructura del mensaje 28
3.8.5 Tramas de datos estándar 29
3.8.6 Tramas de datos extendidas 32
3.8.7 Tramas remotas 32
3.9. BUS CAN Y CANOPEN 33
3.9.1 CAL (CAN Application Layer) 34
3.10. CANOPEN 35
3.10.1 Diccionario de objetos de CANopen 35
3.10.2 Identificadores de mensaje 37

3.10.3 Modelo de comunicaciones *38*
3.10.4 Service Data Objects (SDO) *39*
3.10.5 Process Data Objects (PDO) *42*
3.10.6 Mensajes adicionales *44*
3.10.7 Gestión de la red (NMT) *49*

4. TÉCNICAS Y HERRAMIENTAS 53

4.1. TÉCNICAS METODOLÓGICAS 53
4.1.1 UML (Unified Modeling Language) *53*
4.2. LENGUAJES DE PROGRAMACIÓN 54
4.2.1 C *54*
4.2.2 Java *54*
4.3. ENTORNO DE DESARROLLO 55
4.3.1 SourceBoost IDE *55*
4.3.2 BoostC *56*
4.3.3 Eclipse SDK *56*
4.3.4 Programadora PICkit 2 y adaptador PGM2KIT *57*
4.3.5 232Analyzer *58*
4.3.6 Doxygen *59*
4.3.7 Visual Paradigm for UML Standard Edition 6.0 .... *59*
4.3.8 Microsoft Proyect 2003 .... *59*
4.3.9 COCOMO II *60*
4.4. HARDWARE Y SISTEMA OPERATIVO 60
4.4.1 En el desarrollo del software *60*
4.4.2 Puesta en funcionamiento del sistema *60*

5. ASPECTOS RELEVANTES DEL DESARROLLO 63

5.1. CICLO DE VIDA, ANÁLISIS Y DISEÑO 63
5.2. DESCRIPCIÓN GENERAL SISTEMA 64
5.3. DESCRIPCIÓN DE LOS ELEMENTOS DEL SISTEMA 65
5.4. ARQUITECTURA DEL SISTEMA 66
5.5. CAPAS DE HARDWARE Y DE CONTROL DE HARDWARE 68
5.5.1 Diccionario de objetos de CANopen *69*
5.6. CAPA DE ADAPTACIÓN DE PROTOCOLOS 73
5.7. CAPA DE CONTROL 74
5.7.1 Descripción de la aplicación de control *74*
5.7.2 Breve descripción de la interfaz *75*
5.7.3 Biblioteca CANopen *83*
5.7.4 Internacionalización *85*
5.7.5 Configuración de la interfaz *87*

6. TRABAJOS RELACIONADOS 89

7. CONCLUSIONES 91

8. LÍNEAS DE TRABAJO FUTURAS 93

9. BIBLIOGRAFÍA Y REFERENCIAS 95

Índice de figuras

Figura 1: Carretilla adquirida por el departamento 1
Figura 2: Esquema general de la carretilla 3
Figura 3: Microcontrolador PIC18F258 9
Figura 4: Placa SBC28PC 12
Figura 5: Sensor SRF08 12
Figura 6: Funcionamiento básico de los sensores de ultrasonido 13
Figura 7: Incertidumbre angular en la medida de un ultrasonido 14
Figura 8: Conexiones para el SRF08 15
Figura 9: VScom USB-COM-I 17
Figura 10: Secuencia de inicio en el bus I2C 18
Figura 11: Establecimiento de la comunicación en el bus I2C 19
Figura 12: Secuencia de parada en el bus I2C 19
Figura 13: Direccionamiento en el bus I2C 20
Figura 14: Lectura en el bus I2C 22
Figura 15: Bus CAN 23
Figura 16: Cables del bus CAN 24
Figura 17: Terminadores del bus CAN 24
Figura 18: Controladores del bus CAN 25
Figura 19: Esquema de un nodo en el bus CAN 26
Figura 20: Ejemplo de funcionamiento del bus CAN 28
Figura 21: Estados de la línea en el bus CAN 28
Figura 22: Ejemplo de mensaje CAN 29
Figura 23: Trama de datos estándar del bus CAN 29
Figura 24: Campo de arbitrio de una trama estándar CAN 30
Figura 25: Campo de control de una trama estándar CAN 30
Figura 26: Campo CRC de una trama estándar CAN 31
Figura 27: Campo de confirmación de una trama estándar CAN 31

Figura 28: Trama estándar vs trama extendida en CAN 32
Figura 29: Visión esquemática de los estándares CAN y CANopen en el modelo OSI 33
Figura 30: Estructura del identificador de mensajes CAN 37
Figura 31: Modelo de comunicación productor-consumidor en CANopen 38
Figura 32: Modelos de comunicación punto-a-punto y maestro-esclavo en CANopen............. 38
Figura 33: Parámetros de un objeto SYNC en CANopen 45
Figura 34: Estructura de un mensaje de emergencia en CANopen 45
Figura 35: Trama RTR que el NMT maestro envía a los NMT esclavos.................................. 47
Figura 36: Mensaje que los NMT esclavos envían al NMT maestro 48
Figura 37: Estructura de un mensaje de *Heartbeat* 48
Figura 38: Estructura del mensaje de *Boot-up* 49
Figura 39: Diagrama de transición de estados de un nodo en CANopen 50
Figura 40: Estructura de un mensaje NMT 50
Figura 41: Programadora PICkit 2 57
Figura 42: Adaptador PGM2KIT 57
Figura 43: Conexión a la placa mediante un conector ICSP del tipo ICPC1 58
Figura 44: Esquema de la arquitectura del sistema 61
Figura 45: Ejemplo del sistema en ejecución 62
Figura 46: Ciclo de desarrollo. Proporciona un incremento. 63
Figura 47: Proceso iterativo e incremental 64
Figura 48: Esquema de la arquitectura en capas del sistema 67
Figura 49: Aspecto de la aplicación al arrancar 75
Figura 50: Configuración del puerto serie en Windows 75
Figura 51: Configuración del puerto serie en Linux 76
Figura 52: Comunicación por el puerto serie 76
Figura 53: Mensajes CAN enviados y recibidos 77
Figura 54: Editor de los ficheros de configuración de los nodos 78
Figura 55: Mensajes de error de CANopen 78
Figura 56: Mensajes de alerta de CANopen 79
Figura 57: Ventana principal con un nodo conectado al sistema 79
Figura 58: Controlador para sónares antes de configurar el PIC 80
Figura 59: Configurar la dirección de los sensores 81
Figura 60: Configurar el objeto *PdoEnabled* 81
Figura 61: Configurar el objeto *AlertLimit* 81
Figura 62: Ejemplo del controlador para sónares recibiendo medidas 82
Figura 63: Cambio del estado NMT 83

Figura 64: Cambio de idioma.. 86
Figura 65: Cuadro de diálogo para mensajes CAN en inglés... 87
Figura 66: Mensajes de alerta en inglés 87

Índice de tablas

Tabla 1: Registros del SRF08 15
Tabla 2: Instrucciones del SRF08 16
Tabla 3: Distribución de los COB-IDs en CAL 35
Tabla 4: Estructura de un diccionario de objetos estándar en CANopen 36
Tabla 5: Asignación de los identificadores CAN en CANopen 37
Tabla 6: Códigos de error para SDO *Abort Domain Transfer* 42
Tabla 7: Modos de transmisión de PDOs enCANopen 44
Tabla 8: Códigos de error para los mensajes de emergencia de CANopen 46
Tabla 9: Bits del Error Register de los mensajes de emergencia de CANopen 47
Tabla 10: Valor del campo *state* en un mensaje de NMT *Node Guarding* 48
Tabla 11: Valor del campo *state* en un mensaje de *Heartbeat* 48
Tabla 12: Valores del campo CS del mensaje NMT 51
Tabla 13: Objetos del diccionario comunes para todos los nodos 70
Tabla 14: Objetos del diccionario comunes para los nodos que controlan sensores 71
Tabla 15: Objetos del diccionario comunes para los nodos que controlan actuadores 72
Tabla 16: Objetos del diccionario para los nodos que controlen sensores de ultrasonido 72

1. Introducción

En la actualidad, la automatización de los sistemas logísticos del almacenaje constituye un desafío fundamental que busca optimizar aún más el proceso productivo. Este trabajo se enmarca dentro de un proyecto de investigación más general cuyo principal objetivo es la automatización de una carretilla industrial, con el fin de obtener un sistema autónomo de transporte para realizar tareas de logística y almacenaje. La carretilla (Figura 1) ha sido recientemente adquirida por el Departamento de Informática y Automática, y servirá como plataforma tanto para la prueba de resultados teóricos obtenidos previamente por el grupo de investigación del departamento, como para el desarrollo de posteriores labores de investigación sobre vehículos autónomos y la automatización de una gestión óptima del almacenaje.

Figura 1: Carretilla adquirida por el departamento

El proceso de automatización de la carretilla es necesario plantearlo desde dos aspectos: la arquitectura hardware necesaria para la automatización de los movimientos y la arquitectura software que permita el control de los mismos.

Se deben integrar en la carretilla los elementos físicos necesarios para su movimiento controlado por ordenador, junto con los demás elementos de percepción del entorno por el que se mueve. Además se integrará un ordenador que recogerá las medidas de los sensores y ordenará las acciones a ejecutar. Para esta tarea se desarrollará una arquitectura que permita la interconexión de los diferentes elementos de tal modo que sea fácilmente escalable según aparezcan necesidades futuras referentes al sistema sensorial y de actuación.

En lo relativo a la arquitectura software para el problema del movimiento de forma autónoma, se propone una solución de control que incorpore tareas que van desde el control de los movimientos hasta los comportamientos reactivos para evitar colisiones o seguir caminos. El diseño tendrá en cuenta las siguientes necesidades:

- Una correcta comunicación entre los dispositivos es fundamental para un funcionamiento adecuado sin que se produzcan errores inesperados. Por esta razón se prestará una atención especial a los protocolos de comunicaciones de modo que se puedan detectar errores en la transmisión y existan mecanismos para recuperarse de ellos.
- Los datos provenientes del entorno tendrán diferente naturaleza dependiendo de los sensores de los que se obtenga la información. Por ello, la arquitectura propuesta deberá hacer una integración sensorial con el fin de obtener una información fácilmente manejable y que facilite las tareas de navegación.
- Se debe diseñar una arquitectura flexible que permita realizar cambios en los dispositivos hardware que la integran. Para ello se introducirán capas de abstracción que harán que el acceso a estos dispositivos se haga de una manera similar independientemente de su naturaleza.

En la ilustración “Figura 2” se puede observar el esquema general de todos los elementos que se van a incluir en la carretilla. Este proyecto se centra fundamentalmente en la parte relativa a los sensores y actuadores del sistema. Se plantea desarrollar y llevar a la práctica soluciones arquitectónicas software y hardware para la recogida de los datos sensoriales y el control de los actuadores.

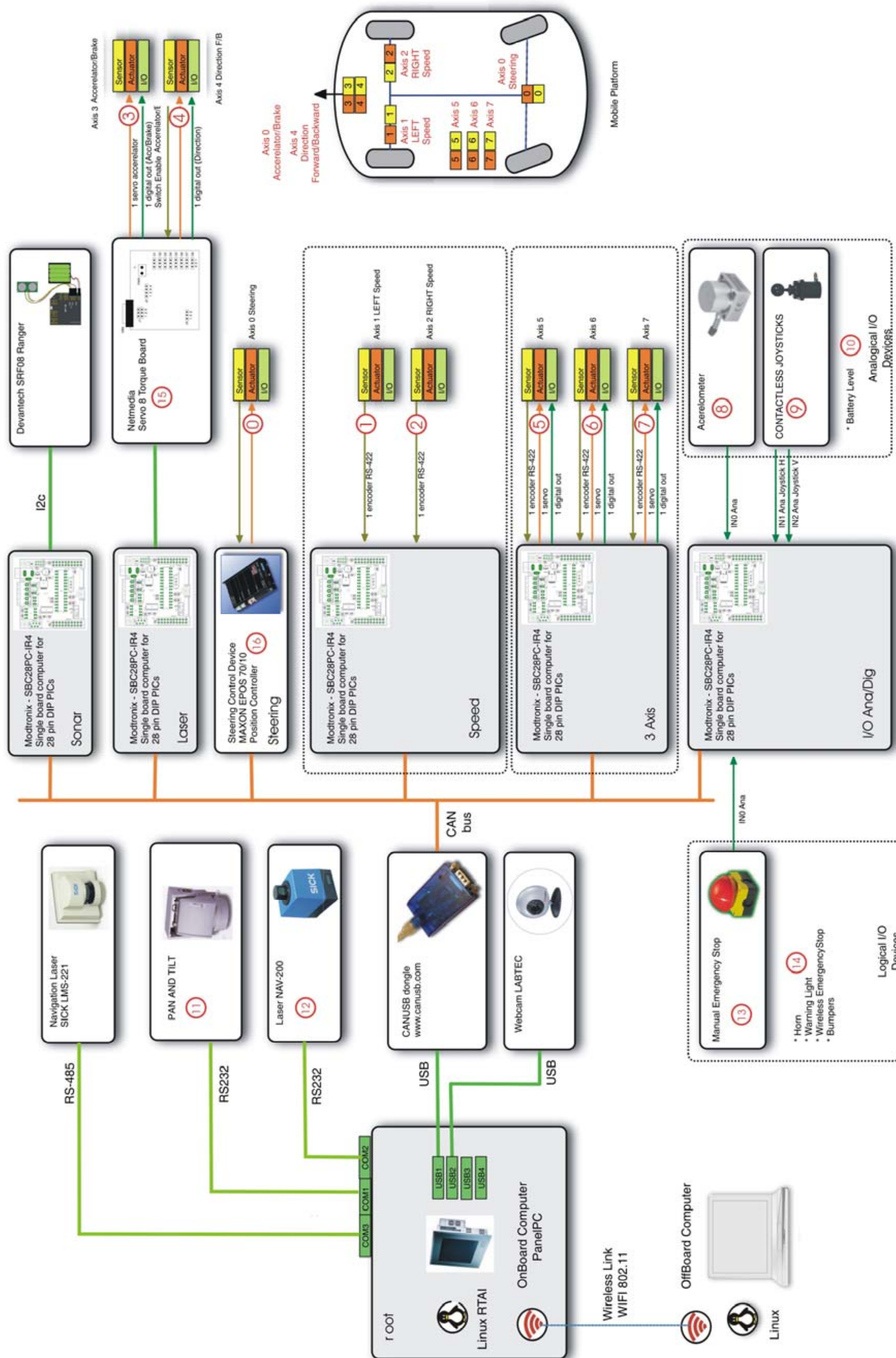


Figura 2: Esquema general de la carretilla

1.1. Breve descripción de contenidos

Esta Memoria ha sido estructurada siguiendo los consejos de la "Guía de realización y documentación para el proyecto final de carrera" [1], y utiliza las normas de formato y estilo de "Normas para la Elaboración de Informes Técnicos" [2]. Consta de los siguientes apartados:

- Apartado 2: **Objetivos**. Enumeración de todos los objetivos que se han perseguido en la elaboración del proyecto.
- Apartado 3: **Conceptos teóricos**. Breve explicación de los sistemas, protocolos e instrumentos utilizados en el proyecto.
- Apartado 4: **Técnicas y herramientas**. Descripción de los lenguajes, metodologías y software utilizados.
- Apartado 5: **Aspectos relevantes del desarrollo**. Descripción de los puntos más interesantes del desarrollo del proyecto.
- Apartado 6: **Trabajos relacionados**. Información sobre el proyecto general en el cual se enmarca este trabajo, y sobre otros trabajos que se están desarrollando en él.
- Apartado 7: **Conclusiones**. Revisión del trabajo realizado y evaluación de los resultados.
- Apartado 8: **Líneas de trabajo futuras**. Definición de objetivos sobre los que se puede seguir trabajando.
- Apartado 9: **Bibliografía y referencias**. Documentación, libros y enlaces de interés.
- Anexo 1: **Plan del proyecto**. Planificación temporal del proyecto. Elaboración de un calendario para su realización.
- Anexo 2: **Especificación de requisitos del software**. Descripción de todos los requisitos que se deben cumplir.
- Anexo 3: **Especificación del diseño**. Explicación de la solución a seguir. Diseño de la arquitectura general del sistema.
- Anexo 4: **Documentación técnica de programación**. Descripción de los subsistemas que conforman el proyecto y documentación de la estructura de los ficheros de código fuente.
- Anexo 5: **Manuales de usuario**. Manuales necesarios para el correcto manejo del sistema y para poder aprovechar toda su funcionalidad y potencia.

1.2. Estructura del CD-ROM

Junto con este documento se ha entregado un CD-ROM con la siguiente estructura de directorios y contenidos:

- **Binarios**: carpeta que contiene todos los ejecutables necesarios para poner en marcha el sistema. En ella están tanto el código en ensamblador para los PICs, como el jar de la interfaz gráfica de usuario.
- **Manuales de usuario**: carpeta con los manuales necesarios para el correcto manejo del sistema.
- **Código fuente**: carpeta que contiene todos los ficheros de código fuente de los diferentes subsistemas.
- **Documentación del proyecto**: carpeta con la documentación de todo el código fuente desarrollado en formato digital. Contiene las APIs documentadas de las bibliotecas de CANopen desarrolladas.
- **Entorno de desarrollo**: carpeta con los instaladores de las principales aplicaciones utilizadas durante el desarrollo del proyecto.
- **Documentación de apoyo**: documentación adicional necesaria para entender correcta y completamente el proyecto. Contiene tanto manuales para manejar las herramientas utilizadas durante el desarrollo, como documentos y guías sobre el protocolo CANopen, el bus CAN, los PICs utilizados y sus placas.
- **Memoria del proyecto**: contiene este documento en formato PDF.

2. Objetivos

2.1. Objetivos del proyecto

El principal objetivo de este proyecto es el de diseñar y desarrollar una arquitectura software y hardware para el control de los sensores y actuadores que irán integrados en una carretilla industrial, la cual se pretende automatizar. Se desarrollará una arquitectura en capas y se definirán las interfaces que cada una ofrece para acceder a sus servicios.

En la carretilla se integrarán distintos sensores y actuadores que permitan realizar un control de sus movimientos. Estos dispositivos estarán controlados mediante microcontroladores de bajo coste, que se comunicarán con ellos por el bus I2C. A su vez los microcontroladores estarán comunicados entre sí mediante el bus de campo CAN.

Los objetivos que debe cumplir el proyecto son:

1. Estudiar las características y funcionamiento del bus CAN, así como realizar una documentación técnica sobre él para que la puedan consultar otros desarrolladores que lo utilicen posteriormente en otros proyectos.

2. Desarrollar software para que los microcontroladores que gestionen sensores cumplan los siguientes requisitos:
   - ✓ Leer continuamente los datos que recogen sus sensores mediante el bus I2C.
   - ✓ Realizar una lectura eficiente de los dispositivos.
   - ✓ Enviar las mediciones por el bus CAN de forma periódica, o bajo demanda.
   - ✓ Enviar alertas si las mediciones obtenidas no se encuentran dentro de los límites establecidos.
   - ✓ Ser capaces de adaptar su comportamiento, es decir, que se puedan configurar (las direcciones de los dispositivos, los temporizadores...) desde una aplicación de control en el PC.
   - ✓ Informar a esa aplicación sobre sus funciones, su estado y el estado de sus dispositivos.

   Si controlan actuadores deberán cumplir los dos últimos objetivos expuestos, y además ser capaces de enviar a sus dispositivos las órdenes que les lleguen por el bus CAN.

3. Desarrollar una interfaz gráfica que permita al usuario manejar y controlar todo el sistema desde el PC. Esta aplicación debe realizar las siguientes funciones:
   - ✓ Detectar automáticamente los microcontroladores conectados al bus CAN.
   - ✓ Recibir los datos tomados por cada sensor.
   - ✓ Enviar órdenes o instrucciones a los actuadores.
   - ✓ Conocer en todo momento el estado de los elementos del sistema: si están funcionando correctamente o no.

- ✓ Ser capaz de configurar los parámetros de los microcontroladores que determinen características de su funcionamiento: las direcciones de los dispositivos, cada cuánto tiempo transmiten los datos de los sensores, cuándo deben enviar alertas, etc.

4. Debido a que el bus CAN sólo define las capas física y de enlace, será necesario hacer uso de algún protocolo que implemente las capas superiores, y especifique cómo se va a realizar la comunicación entre los distintos elementos conectados al bus.

5. Para el correcto funcionamiento del sistema será necesario desarrollar una serie de programas que permitan llevar a cabo la comunicación entre sus diferentes partes. Además éstos deben implementar adecuadamente los protocolos utilizados.

6. Documentar adecuadamente tanto los conceptos teóricos relacionados con el sistema, como las bibliotecas de funciones y programas que se desarrollen, para su reutilización o ampliación en el futuro.

Para una visión más detallada de los objetivos se recomienda consultar el "Anexo 2: Especificación de requisitos del software".

Todos los conceptos introducidos en este apartado están explicados con más detalle en el apartado "3. Conceptos teóricos".

2.2. Objetivos personales

El principal objetivo personal es el de superar el reto que supone el aprender, comprender y poner en práctica todos los conocimientos nuevos necesarios para realizar este proyecto, ya que nunca antes había trabajado con este tipo de sistemas. Me resulta muy interesante la experiencia de programar un microcontrolador por primera vez, y hacer que se comunique mediante los buses I2C y CAN, los cuales nunca he utilizado.

También el hecho de programar a tan bajo nivel, por las dificultades tanto de implementación como de depuración que conlleva, además de tener que aprender las peculiaridades del compilador que voy a utilizar para ello. Espero ser capaz de idear soluciones para resolver todos los problemas que vayan surgiendo durante el desarrollo.

3. Conceptos teóricos

En este apartado haremos una introducción a los conocimientos más básicos que se deben tener para comprender el proyecto. Si se quiere profundizar más en algún concepto hay más información disponible en el directorio "Documentación de apoyo" del CD-ROM adjunto. También se puede consultar la bibliografía propuesta en el capítulo *Bibliografía y referencias*.

3.1. Microcontroladores

Un microcontrolador [43] es un circuito integrado o chip que incluye en su interior las tres unidades funcionales de una computadora: CPU, memoria y unidades de E/S, es decir, se trata de un computador completo en un solo circuito integrado. Aunque sus prestaciones son limitadas, además de dicha integración, su característica principal es su alto nivel de especialización. A pesar de que los hay del tamaño de un sello de correos, lo normal es que sean incluso más pequeños, ya que, lógicamente, forman parte del dispositivo que controlan.

Figura 3: Microcontrolador PIC18F258

Es un microprocesador optimizado para ser utilizado para controlar equipos electrónicos. Los microcontroladores representan la inmensa mayoría de los chips de computadoras vendidos. Sobre un 50% son controladores "simples" y el restante corresponde a DSPs (Procesador digital de señal) [44] más especializados. Por ejemplo, mientras se pueden tener uno o dos microprocesadores de propósito general en casa (un PC), probablemente distribuidos entre los electrodomésticos de un hogar hay una o dos docenas de microcontroladores. Pueden encontrarse en casi cualquier dispositivo eléctrico como automóviles, lavadoras, hornos microondas, teléfonos, etc.

Un microcontrolador difiere de una CPU normal, debido a que es más fácil convertirla en una computadora en funcionamiento, con un mínimo de chips externos de apoyo. La idea es que el chip se coloque en el dispositivo, enganchado a la fuente de energía y de información que necesite, y eso es todo. Un microprocesador tradicional no permite hacer esto, ya que espera que todas estas tareas sean manejadas por otros chips.

Por ejemplo, un microcontrolador típico tendrá un generador de reloj integrado y una pequeña cantidad de memoria RAM y ROM/EPROM/EEPROM, significando que para hacerlo funcionar, todo lo que se necesita son unos pocos programas de control y un cristal de sincronización. Los microcontroladores disponen generalmente también de una gran variedad de dispositivos de entrada/salida, como convertidores de analógico a digital, temporizadores, UARTs y buses de interfaz serie especializados, como I2C y CAN.

Algunos de los microcontroladores más comunes en uso son:

- Microchip [21]:
    - Gama de 8 bits: Familia 10F2XX.
    - Gama baja: Familia 12CXX de 12 bits.
    - Gama media: Familias 12FXX, 16CXX y 16FXX de 14 bits. Por ejemplo el PIC16F84.
    - Gama alta: Familias 18CXX y 18FXX de 16 bits. Por ejemplo el PIC18F258.
    - dsPIC: DSPs.
- Intel [22]:
    - De 8 bits: 8XC42, MCS51, 8XC251.
    - De 16 bits: MCS96, MXS296.
- Freescale (antes Motorola) [23]:
    - De 8 bits: 68HC05, 68HC08, 68HC11.
    - De 16 bits: 68HC12, 68HC16.
    - De 32 bits.
    - 683XX.

3.1.1 PIC18F258

Es un microcontrolador de la familia PIC18F de Microchip. Las especificaciones completas del PIC están en [28]. Algunas de sus características son:

- Frecuencia reloj: 40MHz (10 MIPS).
- Tipo de memoria: Enhanced Flash. Esta tecnología de Microchip ofrece 1.000.000 de ciclos de borrado-escritura, siendo una de las de mayor durabilidad de la industria.
- Tamaño de la memoria flash: 32Kbytes.
- RAM: 1536 *bytes*.
- EEPROM: 256 *bytes*.
- Fácil de programar. Sólo tiene 77 instrucciones sencillas.
- 22 pines de E/S.
- 3 puertos de E/S.
- Tres pines de interrupciones externas.
- Timer0: temporizador/contador de 8/16 bits con *prescaler* programable.
- Timer1: temporizador/contador 16 bits.
- Timer2: temporizador/contador de 8 bits con un registro de periodo de 8 bits.
- Timer3: temporizador/contador de 16 bits.

- Puerto serie síncrono maestro en dos modos:
    - Tres conexiones SPI.
    - I2C maestro y esclavo.
- USART direccionable.
- Conector ICSP (*In-Circuit Serial Programming*). Programación del PIC en el circuito definitivo a usar, sin necesidad de sacarlo.
- Incluye una interfaz para bus CAN 2.0B:
    - Velocidad de hasta 1Mbps.
    - Campos para identificadores de hasta 29 bits.
    - Mensajes de 8 *bytes* de longitud.
    - Tres *buffers* para transmisión de mensajes con prioridades.
    - Dos *buffers* de recepción de mensajes.
    - Seis filtros de aceptación de mensajes con prioridades.
    - Múltiples *buffers* de recepción de mensajes de alta prioridad para prevenir su pérdida por desbordamiento de *buffer*.
    - Avanzadas características para el manejo de errores.

3.2. Placa SBC28PC

La SBC28PC [34] es una placa para microcontroladores PIC de 28 pines (entre ellos el PIC18F258). Posee interfaces para RS485 y bus CAN. Algunas de sus principales características son:

- 22 puertos de E/S.
- Un conector bloque terminal de cinco pines con señales de RS485 o de bus CAN, masa y alimentación.
- La interfaz para RS485 también está disponible por medio de un conector Molex [45] de tres pines.
- Un conector ICSP (*In Circuit Serial Programming*) del tipo ICPC1, por lo que la CPU puede ser programada en el circuito. Para más información sobre la cómo realizar la programación consultar el apartado 4.3.4.

Figura 4: Placa SBC28PC

3.3. Sensor de ultrasonido SRF08

El SRF08 [35] es un sensor de distancias por ultrasonidos para robots que representa una importante generación en sistemas de medidas de distancias por sónar, consiguiendo niveles de precisión y alcance únicos e impensables hasta ahora con esta tecnología. Puede medir hasta 11 metros. El sensor se conecta al microcontrolador mediante un bus I2C (ver apartado 3.6). Con una alimentación única de 5V, sólo requiere 15 mA, para funcionar y 3mA mientras está en reposo. El SRF08 incluye además un medidor de luminosidad mediante LDR [1] que permite conocer el nivel de luz usando igualmente el bus I2C y sin necesidad de recursos adicionales.

Figura 5: Sensor SRF08

[1] Fotorresistencia LDR (Light Dependent Resistor): componentes pasivos cuya resistencia varía en función de la luz que reciben.

3.3.1 Funcionamiento

El ultrasonido es una vibración mecánica con una frecuencia mayor que la máxima que el oído humano es capaz de percibir. El rango de sonidos audibles está entre los 16 y 20 KHz, mientras que los ultrasonidos tienen frecuencias que van desde los 20KHz en adelante.

El funcionamiento básico de los ultrasonidos como medidores de distancia se muestra en el esquema “Figura 6”. En él se tiene un transmisor que emite un pulso de ultrasonido, que rebota en un determinado objeto. La reflexión de ese pulso es detectada por un receptor de ultrasonidos.

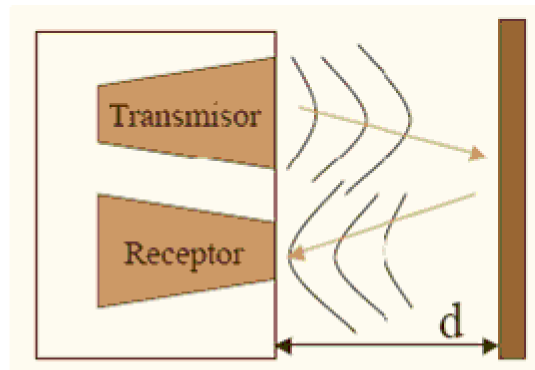


Figura 6: Funcionamiento básico de los sensores de ultrasonido

La mayoría de los sensores de ultrasonido de bajo coste se basan en la emisión de un pulso de ultrasonido cuyo lóbulo, o campo de acción, es de forma cónica. Midiendo el tiempo que transcurre entre la emisión del sonido y la percepción del eco se puede establecer la distancia a la que se encuentra el obstáculo que ha producido la reflexión de la onda sonora, mediante la fórmula:

$$d = \frac{1}{2} V \cdot t$$

donde **V** es la velocidad del sonido en el aire y **t** es el tiempo transcurrido entre la emisión y recepción del pulso.

El principal problema de la toma de medidas con este tipo de sensores es que, como se observa en la siguiente figura, hay cierta incertidumbre sobre la posición exacta del obstáculo que detectan. El campo de actuación del pulso que se emite desde un transductor de ultrasonido tiene forma cónica. El eco que se recibe como respuesta a la reflexión del sonido indica la presencia del objeto más cercano que se encuentra dentro del cono acústico, y no especifica la localización angular del mismo.

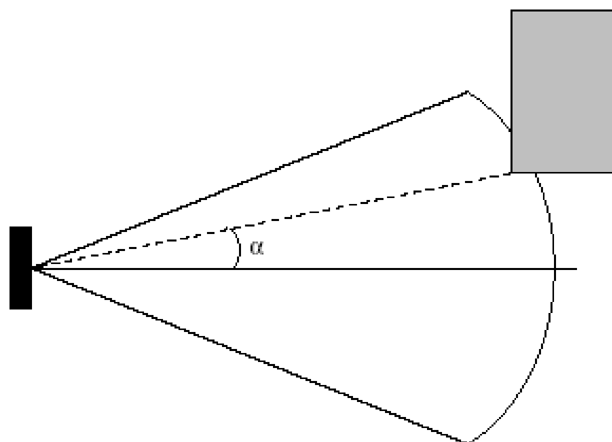


Figura 7: Incertidumbre angular en la medida de un ultrasonido

3.3.2 Direcciones

La dirección por defecto de fábrica del SRF08 es 0xE0. El usuario puede cambiarla por dieciséis direcciones diferentes: E0, E2, E4, E6, E8, EA, EC, EE, F0, F2, F4, F6, F8, FA, FC o FE, por lo que es posible utilizar hasta dieciséis sensores sobre un mismo bus I2C. Además de las direcciones anteriores, todos los sonares conectados al bus I2C responderán a la dirección 0 -al ser la dirección de atención general. Esto significa que escribir un comando de medición de la distancia para la dirección 0x00 de I2C iniciará las mediciones en todos los sensores al mismo tiempo.

3.3.3 Conexiones

El pin señalado como *"Do Not Connect"* (No conectar: NC) debería permanecer sin conexión. Las líneas SCL y SDA deberían tener cada una de ellas una resistencia pull-up de +5v en el bus I2C. Sólo necesita un par de resistencias en todo el bus, no un par por cada módulo o circuito conectado al bus I2C. Normalmente se ubican en el bus maestro en vez de en los buses esclavos. El sensor SRF08 es siempre un bus esclavo - y nunca un bus maestro. Un valor apropiado seria el de 1,8 K en caso de que las necesitase.

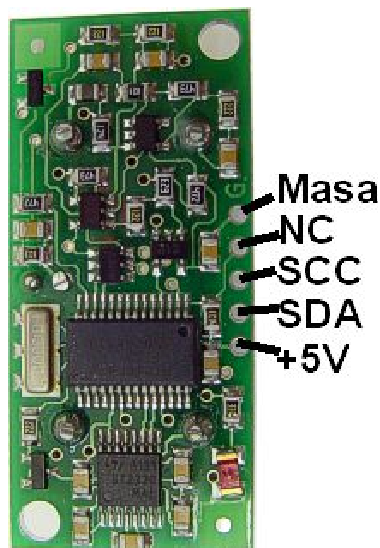


Figura 8: Conexiones para el SRF08

3.3.4 Registros

El sensor SRF08 tiene un conjunto de 36 registros:

Ubicación	Lectura	Escritura
0	Revisión de software	Registro de comando
1	Sensor de luz	Registro de ganancia máx. (por defecto 31)
2	*Byte* alto del 1º eco	Registro de alcance de distancia (por defecto 255)
3	*Byte* bajo del 1º eco	No disponible
----	----	----
34	*Byte* alto del 17º eco	No disponible
35	*Byte* bajo del 17º eco	No disponible

Tabla 1: Registros del SRF08

Solamente se puede escribir en las ubicaciones 0, 1 y 2. La ubicación 0 es el registro de comandos y se utiliza para iniciar la sesión de cálculo de la distancia. No puede leerse. La lectura de la ubicación da como resultado la revisión del software de SRF08. Por defecto, la medición dura 65mS, aunque puede cambiarse modificando el registro de alcance de la ubicación 2. En ese caso hay que cambiar la ganancia analógica en la ubicación 1.

La ubicación 1 es el sensor de luz en placa. Este dato se actualiza cada vez que se ejecuta un comando de medición de distancia y se puede leer cuando se leen los datos de la medición. Las dos ubicaciones siguientes, 2 y 3, son resultados sin signo de 16 bits de la última medición - el nivel lógico alto en primer lugar. El significado de este valor depende del comando utilizado, y puede estar expresado en pulgadas, o en centímetros, o bien el tiempo de vuelo del ping expresado en uS. Un valor cero indica que no se ha detectado objeto alguno. Hay hasta 16 resultados adicionales que indican los ecos de objetos más lejanos.

3.3.5 Instrucciones

Existen tres instrucciones para iniciar una medición de distancia (desde 80 hasta 82), que devuelven el resultado en pulgadas, centímetros o microsegundos. Asimismo, también existe un modo ANN (Artificial Neural Network) y un grupo de instrucciones para modificar la dirección de I2C del srf08. Para más información sobre estas opciones ver [35].

INSTRUCCIONES		ACCIÓN
Decimal	Hexadecimal	
80	0X50	Modo cálculo distancia - Resultado en pulgadas
81	0X51	Modo cálculo distancia - Resultado en centímetros
82	0X52	Modo cálculo distancia - Resultado en microsegundos
83	0X53	Modo ANN - Resultado en pulgadas
84	0X54	Modo ANN - Resultado en centímetros
85	0X55	Modo ANN - Resultado en micro-segundos
160	0XA0	1º en la secuencia para cambiar la dirección I2C
165	0XA5	3º en la secuencia para cambiar la dirección I2C
170	0XAA	2º en la secuencia para cambiar la dirección I2C

Tabla 2: Instrucciones del SRF08

3.3.6 Mediciones

Para iniciar la medición de la distancia, hay que escribir una de las instrucciones anteriores en el registro de comando (registro 0) y esperar el tiempo necesario para la ejecución de la operación. A continuación, deberá leer el resultado en el formato que desee (centímetros, etc). El búfer de eco se pone a cero al comienzo de cada medición. La primera medición del eco se coloca en las ubicaciones 2 y 3, la segunda en 4 y 5, etc. Si una ubicación (niveles altos o bajos de *bytes*) es 0, entonces no se encontrará ningún otro valor en el resto de los registros. El tiempo recomendado y establecido por defecto para realizar la operación es de 65mS, sin embargo es posible acortar este periodo escribiendo en el registro de alcance antes de lanzar la instrucción de medición. Los datos del sensor de luz de la ubicación 1 se actualizarán también después de la instrucción de medición.

3.4. Bus

Un bus [36] conecta lógicamente varios periféricos (o computadores) sobre el mismo conjunto de cables. Se utiliza para la transferencia interna de datos en un sistema computacional en funcionamiento. En un bus todos los nodos reciben los datos aunque no se dirijan a ellos, en cuyo caso simplemente los ignoran.

En los buses modernos se pueden utilizar tanto conexiones paralelas como en serie, y pueden cablearse utilizando diferentes topologías.

3.5. RS485

La norma RS485 [39] ha sido desarrollada para la transmisión de datos en serie, a grandes distancias, a altas velocidades distancias (35 Mbps hasta 10 metros y 100 Kbps en 1200 metros) y reduciendo los ruidos. Es una especificación eléctrica para la capa física del modelo OSI. Está concebida para las comunicaciones como sistema bus bidireccional. El medio físico de transmisión es un par trenzado que admite hasta 32 estaciones en un solo hilo.

3.5.1 VScom USB-COM-I

El VScom USB-COM-I [40] es un conversor de USB a serie. Posee un puerto hembra para RS422/485. Con RS485 la dirección de los datos está controlada por el puerto, sin intervención del software. Admite velocidades de hasta 921,6Kbps. No necesita alimentación externa. Funciona en Windows y Linux.

Figura 9: VScom USB-COM-I

3.6. Bus I2C

El bus I2C [38] es un bus de de comunicaciones serie. Su nombre viene de *Inter-Integrated Circuit* (Circuitos Inter-Integrados). La versión 1.0 data del año 1992 y la versión 2.1 del año 2000. Su diseñador es Philips. La velocidad es de 100Kbits por segundo en el modo estándar, aunque también permite velocidades de 3.4 Mbit/s. Es un bus muy usado en la industria, principalmente para comunicar microcontroladores y sus periféricos en sistemas empotrados (*Embedded Systems*) y generalizando más para comunicar circuitos integrados entre sí que normalmente residen en un mismo circuito impreso. Para información más detallada sobre las especificaciones de este bus consultar [29].

3.6.1 Características principales

La principal característica del I2C es que sólo usa dos hilos para transmitir la información: por uno van los datos y por otro la señal de reloj que sirve para sincronizarlos. También es necesaria una tercera línea, pero esta sólo es la referencia (masa). Como suelen comunicarse circuitos en una misma placa que comparten una misma masa esta tercera línea no suele ser necesaria. Las líneas son:

- **SCL** (System Clock): es la línea de los pulsos de reloj que sincronizan el sistema.
- **SDA** (System Data): es la línea por la que se transmiten los datos entre los dispositivos.
- **GND** (Ground): masa común.

Los dispositivos conectados al bus I2C tienen una dirección de siete bits única para cada uno. Esto significa que puede haber conectados hasta 128 dispositivos. Pueden ser maestros o esclavos. El dispositivo maestro siempre es el que inicia la transferencia de datos, y además genera la señal de reloj. No es necesario que el maestro sea siempre el mismo dispositivo. Esto hace que al bus I2C se le denomine bus multimaestro. No es posible la comunicación directa entre maestros o entre esclavos.

3.6.2 Protocolo de comunicaciones

Para que el maestro pueda iniciar la comunicación el bus debe de estar libre. Esto significa que el las líneas SDA y SCL deben de estar inactivas, presentando un estado lógico alto. En este estado cualquier dispositivo maestro puede ocupar el bus, estableciendo la condición de inicio (START). Esta condición se presenta cuando un dispositivo maestro pone en estado bajo la línea de datos (SDA), pero dejando en alto la línea de reloj (SCL).

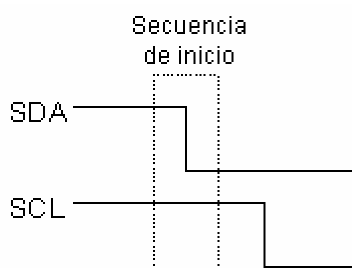


Figura 10: Secuencia de inicio en el bus I2C

El primer *byte* que se transmite después de la condición de inicio contiene siete bits que componen la dirección del dispositivo que se desea seleccionar, y un octavo bit que corresponde a la operación que se quiere realizar con él (lectura o escritura).

Si el dispositivo cuya dirección corresponde a la que se indica en los siete bits (A0-A6) está presente en el bus, éste contesta con un bit en bajo, ubicado inmediatamente después del octavo bit que ha enviado el dispositivo maestro. Este bit de confirmación (ACK) en bajo le indica al dispositivo maestro que el esclavo reconoce la solicitud y está en condiciones de comunicarse. En este momento la comunicación queda establecida y comienza el intercambio de información entre los dispositivos.

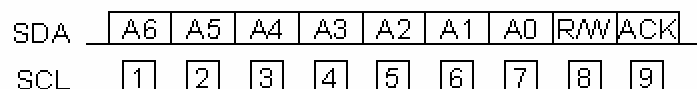


Figura 11: Establecimiento de la comunicación en el bus I2C

Si el bit de lectura/escritura (R/W) fue puesto en esta comunicación a nivel lógico bajo (escritura), el dispositivo maestro envía datos al dispositivo esclavo. Esto se mantiene mientras continúe recibiendo señales de confirmación, y el contacto concluye cuando se hayan transmitido todos los datos.

En el caso contrario, cuando el bit de lectura/escritura estaba a nivel lógico alto (lectura), el dispositivo maestro genera pulsos de reloj para que el dispositivo esclavo pueda enviar los datos. Después de cada *byte* recibido, el dispositivo maestro (a quien le envían los datos) genera un pulso de confirmación (ACK).

El dispositivo maestro puede dejar libre el bus generando una condición de parada (STOP). En ese caso la línea de datos y la de reloj toman un estado lógico alto.

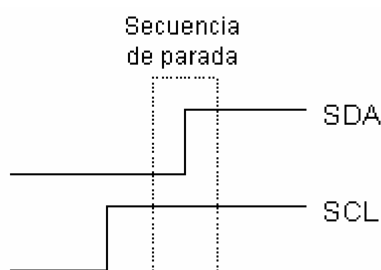


Figura 12: Secuencia de parada en el bus I2C

Si se desea seguir transmitiendo, el dispositivo maestro puede generar otra condición de inicio en lugar de una condición de parada. Esta nueva condición de inicio se denomina "inicio reiterado" y se puede emplear para direccionar un dispositivo esclavo diferente o para alterar el estado del bit de lectura/escritura.

Durante la comunicación los datos se transfieren en secuencias de ocho bits. Estos bits se colocan en la línea SDA comenzando por el bit más significativo. Una vez puesto un bit en SDA, se lleva la línea SCL a alto. Por cada ocho bits que se transfieren, el dispositivo que recibe el dato envía de regreso un bit de confirmación, de modo que en realidad por cada *byte* de datos se producen nueve pulsos sobre la línea de reloj (SCL). Si el dispositivo que recibe envía un bit de confirmación bajo, indica que ha recibido el dato y que está listo para aceptar otro *byte*. Si devuelve un alto, indica que no puede recibir más datos y por lo tanto el dispositivo maestro debería terminar la transferencia enviando una secuencia de parada.

3.6.3 Direccionamiento en el bus I2C

El bus I2C utiliza una palabra de 8 bits para direccionar los dispositivos. Existe un nuevo estándar que utiliza 10 bits, pero está poco difundido. Los dispositivos maestros no tienen dirección y los esclavos se dirigen al dispositivo maestro que les a interrogado, depositando simplemente el valor en el bus.

Cada dispositivo tiene su propia dirección de siete bits. De ellos cuatro son una parte fija, establecida por el fabricante. Los otros tres son la parte variable, modificable directamente por el usuario. Por esta razón en el mismo bus sólo podrán coexistir ocho dispositivos iguales.

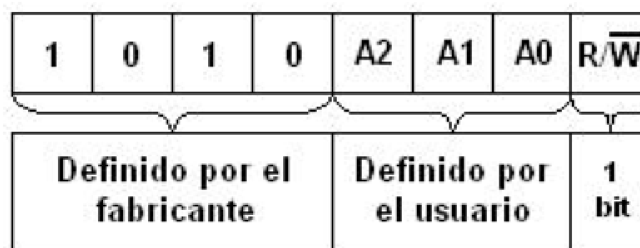


Figura 13: Direccionamiento en el bus I2C

3.6.4 Escritura en un dispositivo esclavo

Cuando el maestro quiere realizar una escritura en un dispositivo esclavo se siguen los siguientes pasos:

1. Enviar una secuencia de inicio.
2. Enviar la dirección de dispositivo con el bit de lectura/escritura en bajo.
3. Enviar el número de registro interno en el que se desea escribir.
4. Enviar el *byte* de datos.
5. Opcionalmente, enviar más *bytes* de datos.
6. Enviar la secuencia de parada.

Como ejemplo, veamos cómo se realizaría una escritura sobre un sónar del modelo SRF08, que tiene una dirección de bus fijada en fábrica de 0xE0. Para comenzar una medición de distancia con el SRF08 se debe escribir 0x51 en el registro de comando, ubicado en la dirección interna 0x00. La secuencia es la que sigue:

1. Enviar una secuencia de inicio.
2. Enviar 0xE0 (La dirección de dispositivo del SRF08 con el bit de lectura/escritura en bajo => Escritura).
3. Enviar 0x00 (dirección interna del registro de comando).
4. Enviar 0x51 (la instrucción para comenzar la medición del SRF08).
5. Enviar la secuencia de parada.

3.6.5 Lectura desde un dispositivo esclavo

Esta operación es algo más complicada. Antes de leer datos desde el dispositivo esclavo, primero se le debe informar desde cuál de sus direcciones internas se va a leer, de manera que una lectura desde un dispositivo esclavo en realidad comienza con una operación de escritura en él.

Se envía la secuencia de inicio, la dirección de dispositivo con el bit de lectura/escritura en bajo y el registro interno desde el que se desea leer. A continuación se envía otra secuencia de inicio con la dirección de dispositivo, pero esta vez con el bit de lectura/escritura en alto. Después se leen todos los *bytes* necesarios y se termina la transacción con una secuencia de parada.

Si por ejemplo quisiéramos recuperar los datos que ha tomado el sensor sónar SRF08 tenemos que leer tres registros: el 0x01 que contiene la luz, el 0x02 que contiene la el *byte* alto de la medición, y el 0x03 que contiene el *byte* bajo. Seguiríamos el siguiente procedimiento:

1. Enviar la secuencia de inicio.
2. Enviar 0xE0 (La dirección de dispositivo del SRF08 con el bit de lectura/escritura en bajo => Escritura).
3. Enviar 0x01 (dirección interna del registro de luz, que es el primero de los que queremos leer).
4. Enviar una secuencia de reinicio.
5. Enviar 0xE1 (La dirección de dispositivo del SRF08 con el bit de lectura/escritura en alto => Lectura).
6. Leer el primer *byte* de datos, que se corresponde con el registro de luz, y enviar una secuencia de reconocimiento (ACK). Al enviar el ACK el dispositivo continuará enviando los datos de los registros consecutivos, con lo cual podremos obtener el valor de la medición tomada por el sensor.
7. Leemos el *byte* alto de la medición y enviamos ACK.
8. Leemos el *byte* bajo de la medición y pero no confirmamos el *byte* (enviamos NACK), para indicar el final de la comunicación.
9. Enviar la secuencia de parada.

En la siguiente figura podemos un ejemplo de cómo se realizaría una lectura en el bus I2C, si la dirección del dispositivo es 0xC0:

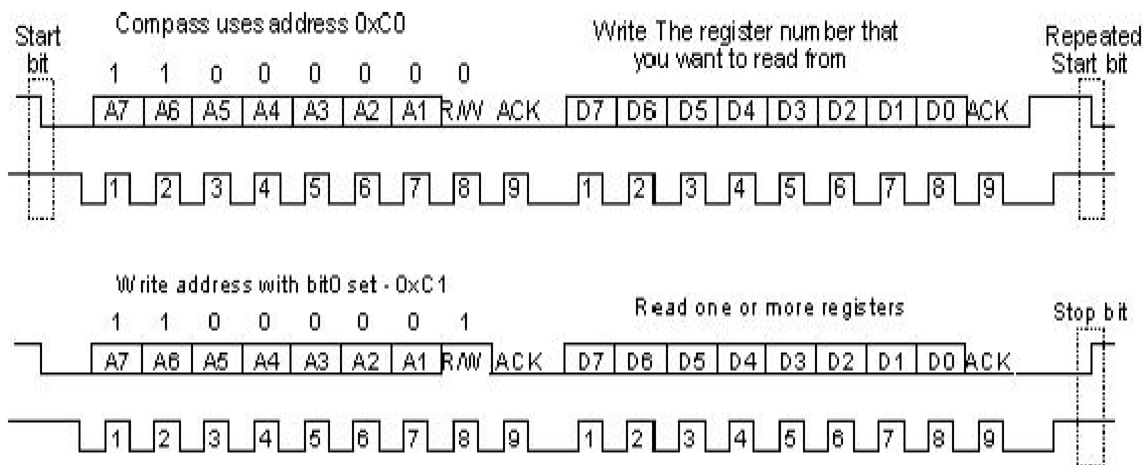


Figura 14: Lectura en el bus I2C

3.7. Bus de campo

Un bus de campo [37] es un sistema de comunicaciones industrial para el control distribuido en tiempo real. Se utiliza para intercambiar datos entre sistemas de automatización y dispositivos de campo. El objetivo de un bus de campo es sustituir las conexiones punto a punto entre los elementos de campo y el equipo de control a través del tradicional bucle de corriente de 4-20mA.

Típicamente son redes digitales, bidireccionales, multipunto, montadas sobre un bus serie, que conectan dispositivos de campo como PLCs (*Programmable logic controllers*), transductores, actuadores y sensores. Cada dispositivo de campo incorpora cierta capacidad de proceso, que lo convierte en un dispositivo inteligente, manteniendo siempre un costo bajo. Cada uno de estos elementos será capaz de ejecutar funciones simples de diagnóstico, control o mantenimiento, así como de comunicarse bidireccionalmente a través del bus.

Un sistema automatizado de control industrial normalmente necesita una jerarquía de sistemas de control para funcionar. En la raíz de esta jerarquía suele encontrarse una máquina de interfaz humana (HMI). Normalmente esto está unido a una capa intermedia de PLCs por un sistema de comunicaciones que no es en tiempo real (por ejemplo Ethernet). Al final de esta cadena de control está el bus de campo, que une los PLCs con los componentes que realmente hacen el trabajo como por ejemplo: sensores, actuadores, motores eléctricos, etc.

3.8. Bus CAN

CAN (*Controller Area Network*) [31] es un protocolo de comunicaciones en serie basado en una topología bus para la transmisión de información entre múltiples unidades centrales de proceso. Es un bus de campo. Originalmente fue concebido para aplicaciones en el área automotriz, pero rápidamente despertó una creciente atención en el área de control y automatización industrial.

Sus principales ventajas son:

- Al ser un protocolo de comunicaciones normalizado se simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre una red común o bus.
- El procesador anfitrión (*host*) delega la carga de comunicaciones a un periférico inteligente, por lo tanto el procesador anfitrión dispone de mayor tiempo para ejecutar sus propias tareas.
- Al ser una red multiplexada reduce considerablemente el cableado y elimina las conexiones punto a punto. También se reduce el número de sensores utilizados ya que permite compartir una gran cantidad de información entre las unidades de control abonadas al sistema. Si por ejemplo una unidad de mando dispone de una información, como la temperatura del motor, la transmite por el bus y de esta forma le llega a todas la unidades que la necesiten.
- Las funciones son repartidas entre distintas unidades de control, por lo que incrementar la funcionalidad no presupone un coste adicional excesivo.

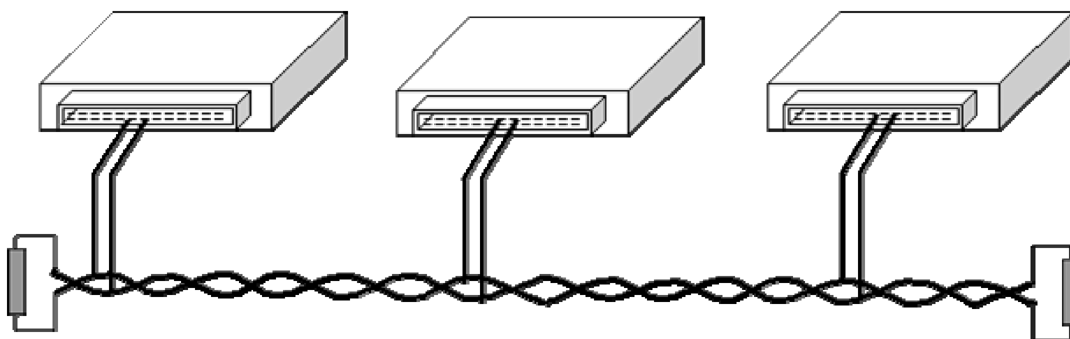


Figura 15: Bus CAN

CAN es un protocolo orientado a mensajes, es decir, la información que se va a intercambiar se descompone en mensajes que se encapsulan en tramas para su transmisión. A cada mensaje se le asigna un identificador único dentro de la red, con el cual los nodos pueden decidir si aceptarlo o no.

3.8.1 Componentes del sistema Can-Bus

Cables

No se ha especificado el medio de transmisión pero lo más común es utilizar dos cables trenzados (para anular los campos magnéticos) que unen todas las unidades de control que forman el sistema. Esta información se trasmite por diferencia de tensión entre los dos cables, de forma que un valor alto de tensión representa un "1" lógico y un valor bajo de tensión representa un "0". La combinación adecuada de unos y ceros conforman el mensaje a trasmitir.

En un cable los valores de tensión oscilan entre 0-2,25V, por lo que se denomina cable L (*Low*) y en el otro, el cable H (*High*) lo hacen entre 2,75-5V. En caso de que se interrumpa la línea H o que se derive a masa, el sistema trabajará con la señal de *Low* con respecto a masa, en el caso de que se interrumpa la línea L, ocurrirá lo contrario. Esta situación permite que el sistema siga trabajando con uno de los cables cortados o comunicados a masa, incluso con ambos comunicados también sería posible el funcionamiento, quedando fuera de servicio solamente cuando ambos cables se cortan.

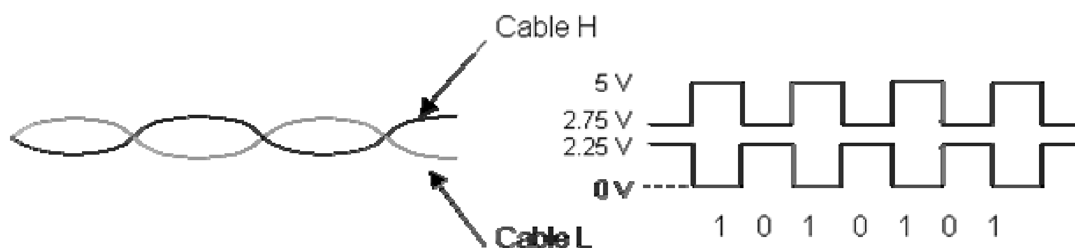


Figura 16: Cables del bus CAN

Elemento de cierre o terminador

Son resistencias conectadas a los extremos de los cables H y L. Normalmente van alojadas en el interior de las unidades de control. Sus valores se obtienen de forma empírica y permiten adecuar el funcionamiento del sistema a diferentes longitudes de cables y número de unidades de control abonadas, ya que impiden fenómenos de reflexión que pueden perturbar el mensaje.

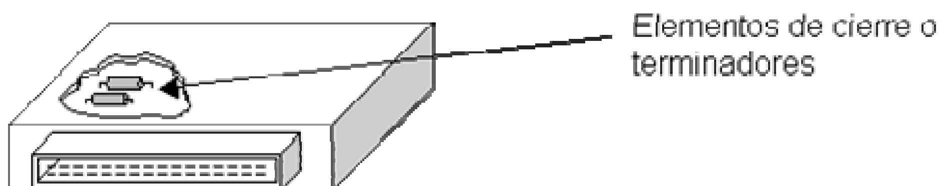


Figura 17: Terminadores del bus CAN

Controlador

Elemento situado en cada unidad de control encargado de la comunicación entre el microprocesador de la unidad de control y el trasmisor-receptor. Sus principales funciones son:

- Acondicionar la información que entra y sale entre el microprocesador y el transmisor-receptor
- Determinar la velocidad de transmisión de los mensajes, que será más o menos elevada según el compromiso del sistema.
- También interviene en la sincronización entre las diferentes unidades de mando para la correcta emisión y recepción de los mensajes.

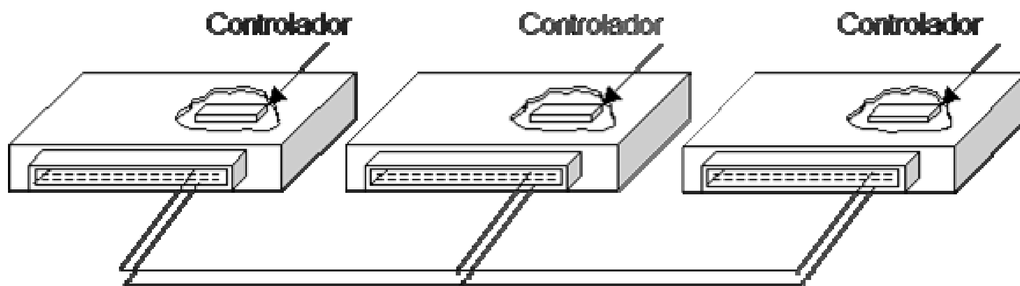


Figura 18: Controladores del bus CAN

Transmisor-receptor

Es el elemento encargado de recibir y trasmitir los datos. Prepara la información para que pueda ser manejada por los controladores, pero en ningún caso interviene modificando el contenido del mensaje. Esta preparación consiste en situar los niveles de tensión de forma adecuada, amplificando la señal cuando la información se vuelca en la línea y reduciéndola cuando es recogida de la misma y suministrada al controlador.

El trasmisor-receptor es básicamente un circuito integrado que está situado en cada una de las unidades de control abonadas al sistema, entre los cables que forman la línea Can-Bus y el controlador.

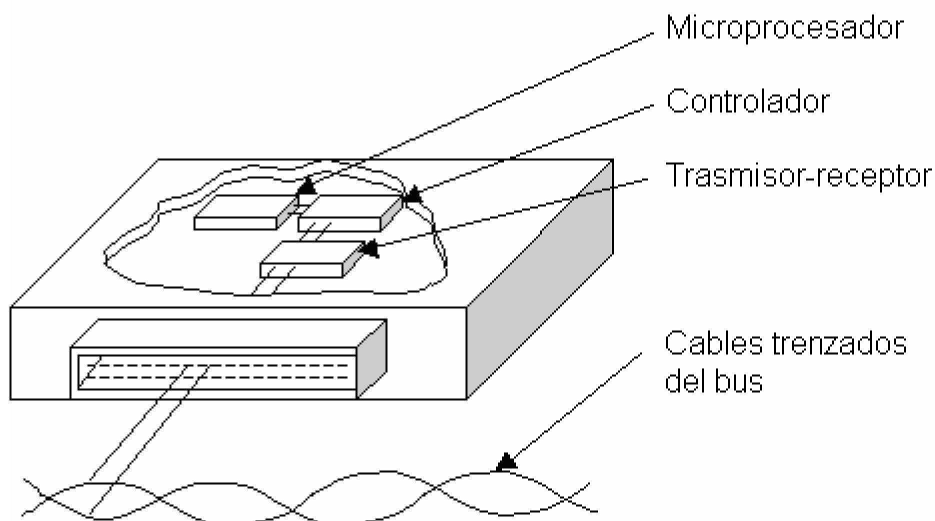


Figura 19: Esquema de un nodo en el bus CAN

3.8.2 Principales características del protocolo CAN

CAN es un protocolo de comunicaciones serie que soporta el control distribuido en tiempo real con un alto nivel de seguridad y multiplexación.

La información que circula entre las unidades de control a través de los dos cables del bus son paquetes de *bits* ("0" y "1") con una longitud limitada y con una estructura definida de campos que conforman el mensaje.

Uno de esos campos actúa de identificador del tipo de dato que se transporta, unidad de control que lo trasmite y prioridad de transmisión. El mensaje no va direccionado a ninguna unidad de control en concreto, sino que cada una de ellas reconocerá mediante este identificador si el mensaje le interesa o no.

Todas las unidades de control son trasmisoras y receptoras, y puede haber un número variable de éstas en el sistema.

Una unidad de control puede solicitar a otra una determinada información mediante uno de los campos del mensaje (trama remota o RTR).

Cualquier unidad de control introduce un mensaje en el bus con la condición de que éste esté libre. Si otra lo intenta al mismo tiempo el conflicto se resuelve por la prioridad del mensaje, indicada en el identificador del mismo.

El sistema está dotado de una serie de mecanismos que aseguran que el mensaje es trasmitido y recibido correctamente. Cuando un mensaje presenta un error es anulado y vuelto a trasmitir de forma correcta. Si hay una unidad de control con problemas, ésta queda fuera de servicio y el sistema sigue funcionando.

Se puede trabajar con diferentes velocidades de transmisión, llegando a alcanzar incluso 1Mbps.

3.8.3 Funcionamiento del Can-Bus

El sistema Can-Bus está orientado al mensaje y no al destinatario. La información en la línea es trasmitida en forma de mensajes estructurados en la que una parte del mismo es el identificador, que indica la clase de dato que contiene. Todas las unidades de control reciben el mensaje, lo filtran y sólo lo emplean las que necesitan dicho dato.

Cada una de ellas puede tanto introducir como recoger mensajes de la línea, y cuando el bus está libre, empezar a trasmitir un nuevo mensaje.

El método de acceso al medio utilizado es el e Acceso Múltiple por Detección de Portadora, con Detección de Colisiones y Arbitraje por Prioridad de Mensaje (CSMA/CD+AMP). Según este método, cuando una unidad desee transmitir debe esperar a que el bus esté libre (detección de portadora). Si esto se cumple transmite un *bit* de inicio (acceso múltiple). Lee el bus durante la transmisión de la trama y compara el *bit* enviado con el recibido. Mientras los *bits* sean idénticos, continúa transmitiendo. Si en algún momento son diferentes, se lleva a cabo el mecanismo de arbitraje.

Cuando varios nodos tratan de transmitir simultáneamente se produce una colisión. Esta situación se resuelve asignando prioridades a los mensajes mediante el identificador. La asignación se realiza durante el diseño del sistema en forma de números binarios y no puede modificarse dinámicamente. El identificador con menor número binario es el que tiene mayor prioridad.

Cuando una unidad de control necesita cierta información envía un mensaje de petición de datos. La unidad que pueda aportar dicha información envía entonces un mensaje con los datos. Estos dos mensajes llevan el mismo identificador. Por lo tanto, en caso de que colisionen, siempre tendrá mayor prioridad el de envío de datos que el de petición.

El proceso de transmisión de datos se desarrolla siguiendo un ciclo de varias fases:

- **Suministro de datos**: Una unidad de control recibe información de los sensores que tiene asociados (por ejemplo: rpm del motor, velocidad, temperatura del motor, puerta abierta, etc.). Su microprocesador pasa la información al controlador, donde es gestionada y acondicionada para que pueda ser manejada por el trasmisor-receptor, donde se transforma en señales eléctricas.
- **Transmisión de datos**: El controlador de dicha unidad transfiere los datos y su identificador junto con la petición de inicio de transmisión, asumiendo la responsabilidad de que el mensaje sea correctamente trasmitido a todas las unidades de control asociadas. Como ya se ha explicado anteriormente, para trasmitir el mensaje el bus debe estar libre y en caso de colisión, el mensaje debe tener una prioridad mayor. Si se cumplen estas condiciones y comienza la transmisión, el resto de unidades de control se convierten en receptoras.
- **Recepción del mensaje**: Cuando todas las unidades de control reciben el mensaje, verifican el identificador para determinar su tipo. Las que necesiten los datos del mensaje lo procesan, y las que no lo ignoran.

El sistema Can-Bus dispone de mecanismos para detectar errores en la transmisión. Las unidades emisoras realizan ciertos mecanismos de control, y todos los receptores comprueban la integridad del mensaje analizando su campo CRC. Estas y otras medidas hacen que las probabilidades de error en la emisión y recepción de mensajes sean muy bajas, por lo que es un sistema muy seguro.

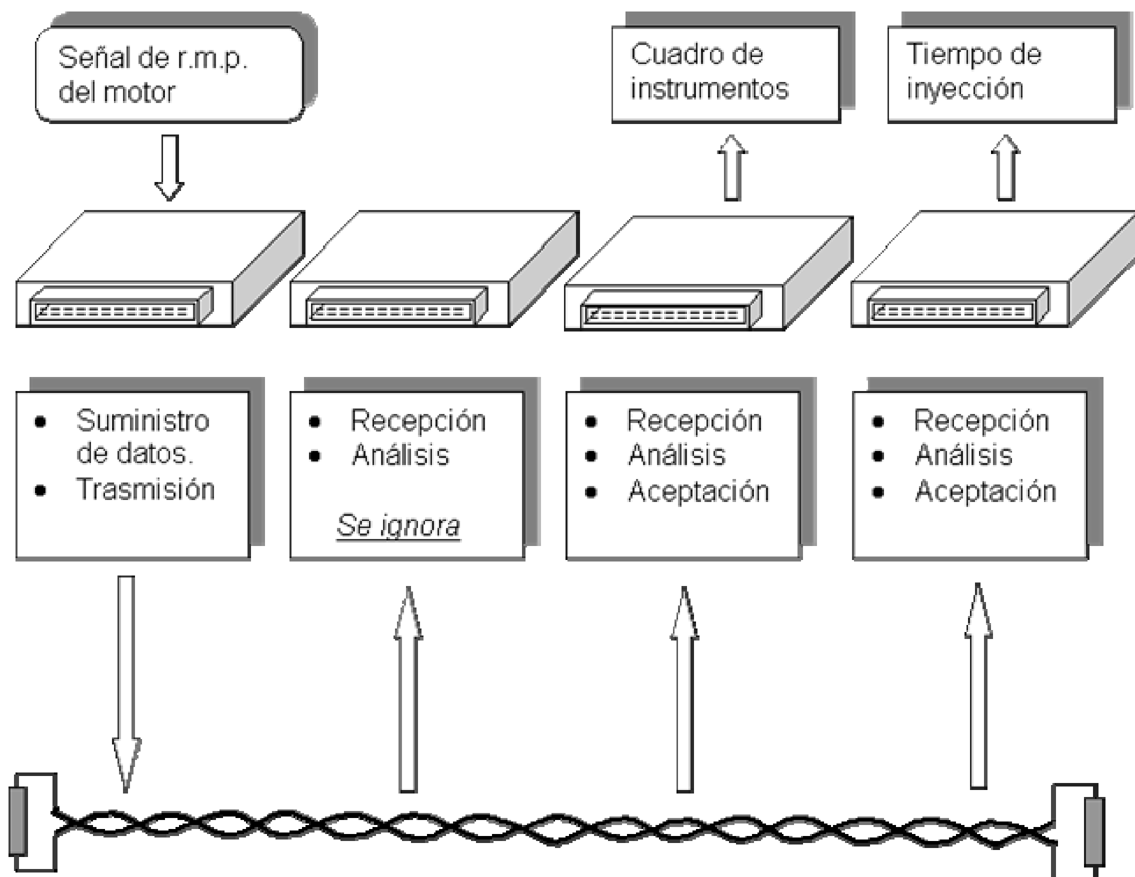


Figura 20: Ejemplo de funcionamiento del bus CAN

3.8.4 Estructura del mensaje

El bus puede tener uno de los dos valores lógicos complementarios: dominante (“0” lógico) o recesivo (“1” lógico). Un mensaje (o trama) es una sucesión de “0” y “1” que se denominan *bits* y se representan por diferentes niveles de tensión en los cables del Can-Bus.

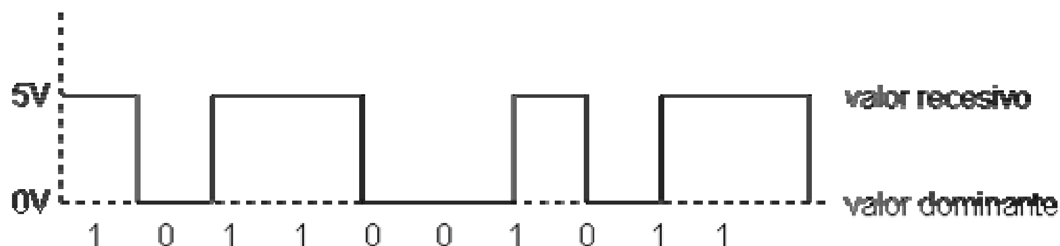


Figura 21: Estados de la línea en el bus CAN

Puede ocurrir que en determinados mensajes se produzcan largas cadenas de ceros o unos, y que esto provoque una pérdida de sincronización entre unidades de mando. El protocolo CAN resuelve esta situación insertando un *bit* de diferente polaridad cada cinco *bits* iguales: cada cinco "0" se inserta un "1" y viceversa. La unidad de mando que recibe el mensaje, descarta el *bit* siguiente a cinco *bits* iguales. Estos bits reciben el nombre de *bit stuffing*.

Ejemplo de un mensaje real:

SOF	identificador	RTR	DE	DLC	DATO byte 1	DATO byte 2	CRC	ACK	FN
0	1100010000	0	000	0010	00010110	00000000	0	01	11111

Figura 22: Ejemplo de mensaje CAN

Existen cuatro tipos de mensajes:

- Trama de datos: transporta los datos. Según la longitud del identificador son estándar (11 *bits*) o extendida (29 *bits*).
- Trama remota: realiza una petición de datos.
- Trama de error: transmitida por una unidad de control cuando detecta un error en el bus.
- Trama de sobrecarga: utilizada para introducir retrasos en la red cuando por ejemplo se necesita un tiempo de espera entre tramas de petición-respuesta.

3.8.5 Tramas de datos estándar

La siguiente figura muestra la estructura de una trama de datos estándar:

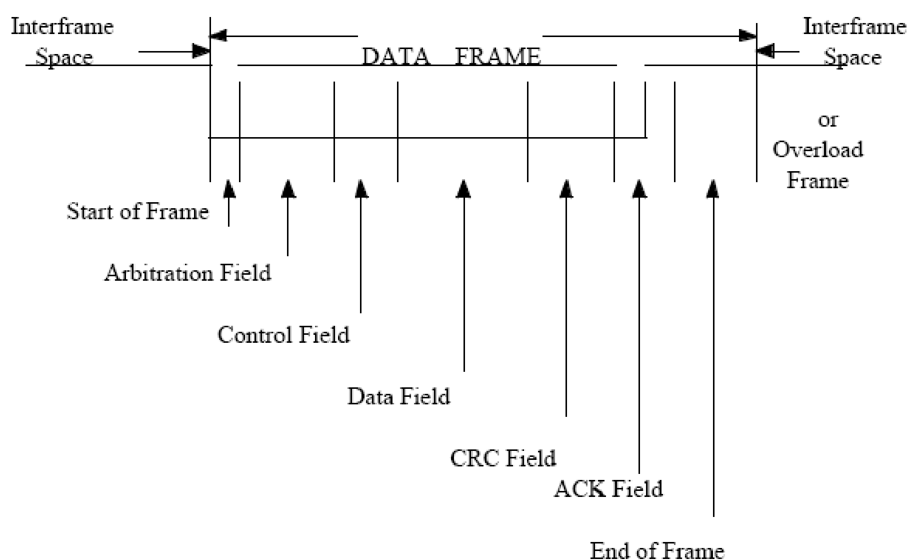


Figura 23: Trama de datos estándar del bus CAN

Están formadas por siete campos:

- **Inicio de trama**: Señala el inicio de una trama de datos o remota. Es un *bit* dominante. Su flanco descendente es utilizado por las unidades para sincronizarse entre sí.
- **Campo de arbitrio**: está formado por el identificador y el *bit* RTR.
    - Identificador: Tiene una longitud de 11bits. Se utiliza para dar prioridad a los mensajes. Cuanto más bajo es el identificador, más prioridad tiene. Y si los identificadores son iguales, la trama de datos tiene más prioridad que la remota. Los *bits* se transmiten comenzando por el más significativo (del ID-10 al ID-0). Los *bits* del ID-10 al ID-4 no pueden ser todos recesivos.
    - *Bit* RTR: Indica si es una trama de datos (RTR=0) o remota (RTR=1).

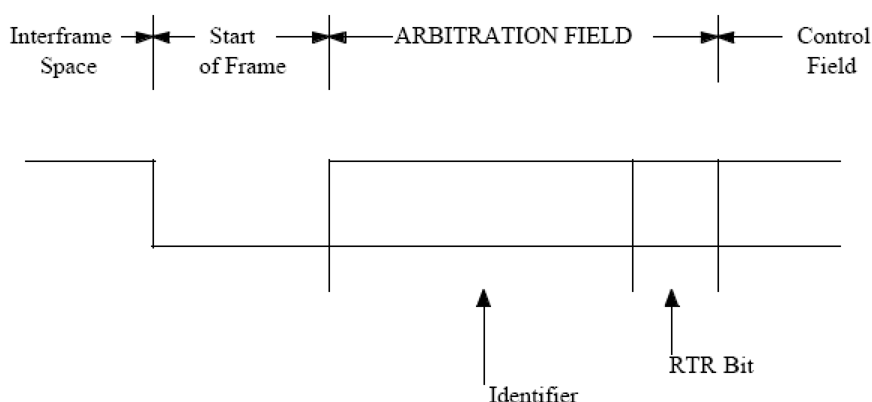


Figura 24: Campo de arbitrio de una trama estándar CAN

- **Campo de control**: Informa sobre las características del campo de datos. Tiene seis *bits*: dos reservados (r1 y r0) y cuatro para el campo de longitud de datos.
    - Campo de longitud de datos: Indica el número de *bytes* del campo de datos.

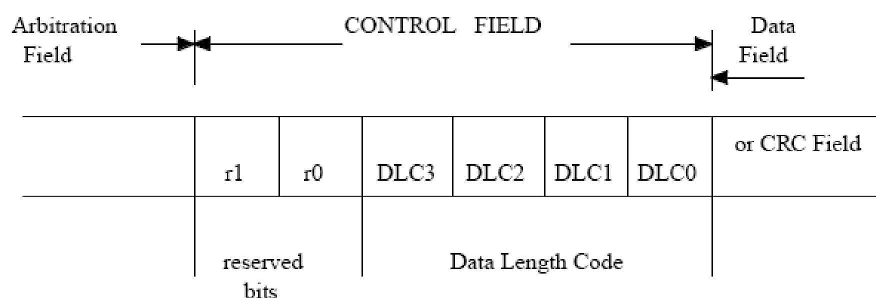


Figura 25: Campo de control de una trama estándar CAN

- **Campo de datos**: Son los datos transferidos en una trama de datos. Puede tener una longitud de entre cero y ocho *bytes*.
- **Campo CRC**: Está formado por la secuencia CRC (código de redundancia cíclica) seguida del delimitador de CRC.
    - Secuencia de CRC: Se utiliza para la detección de errores en la trama. Tiene una longitud de 15 *bits*.
    - Delimitador de CRC: Siempre es un *bit* recesivo ("1").

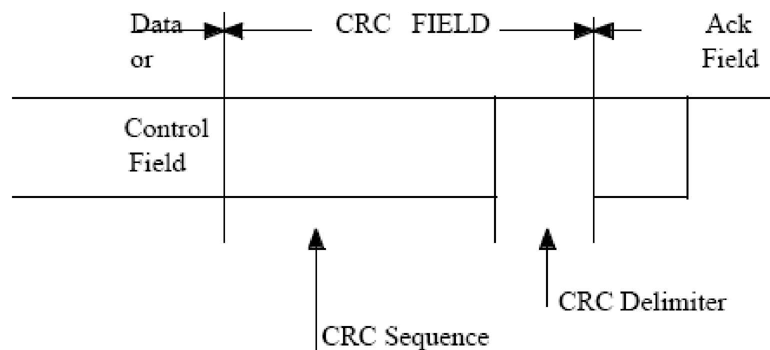


Figura 26: Campo CRC de una trama estándar CAN

- **Campo de confirmación**: Está compuesto por dos bits: ACK y delimitador de ACK. Se utiliza para asentir las tramas. La unidad transmisora siempre transmite ambos como recesivos. Cuando otra unidad recibe la trama correctamente cambia el *bit* de ACK por un *bit* dominante. De esta manera avisa a la unidad emisora de que el mensaje se ha recibido sin errores.

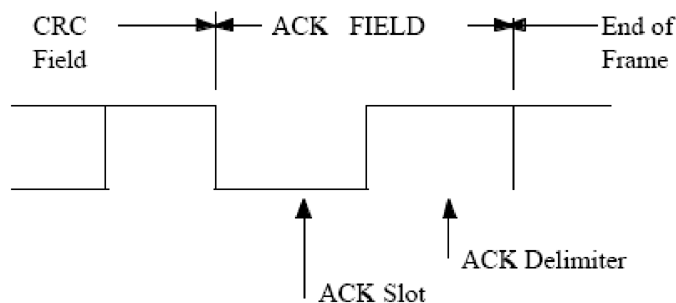


Figura 27: Campo de confirmación de una trama estándar CAN

- **Campo de fin de trama**: Tanto las tramas de datos como las recesivas están delimitadas por una secuencia de siete *bits* recesivos.

3.8.6 Tramas de datos extendidas

Son iguales que las anteriores excepto el identificador:

- Para las tramas de datos estándar tiene una longitud de 11 *bits* que se corresponden con el campo de identificador base explicado anteriormente.
- Para las tramas de datos extendidas tiene una longitud de 29 *bits* y se compone de dos secciones: el campo de identificador base (11 *bits*) y el extendido (18 *bits*).

Para distinguirlas se utiliza el *bit* reservado r1 del campo de control. Se le denomina *bit* IDE. Cuando es un "0" indica que es una trama estándar y cuando es un "1" extendida.

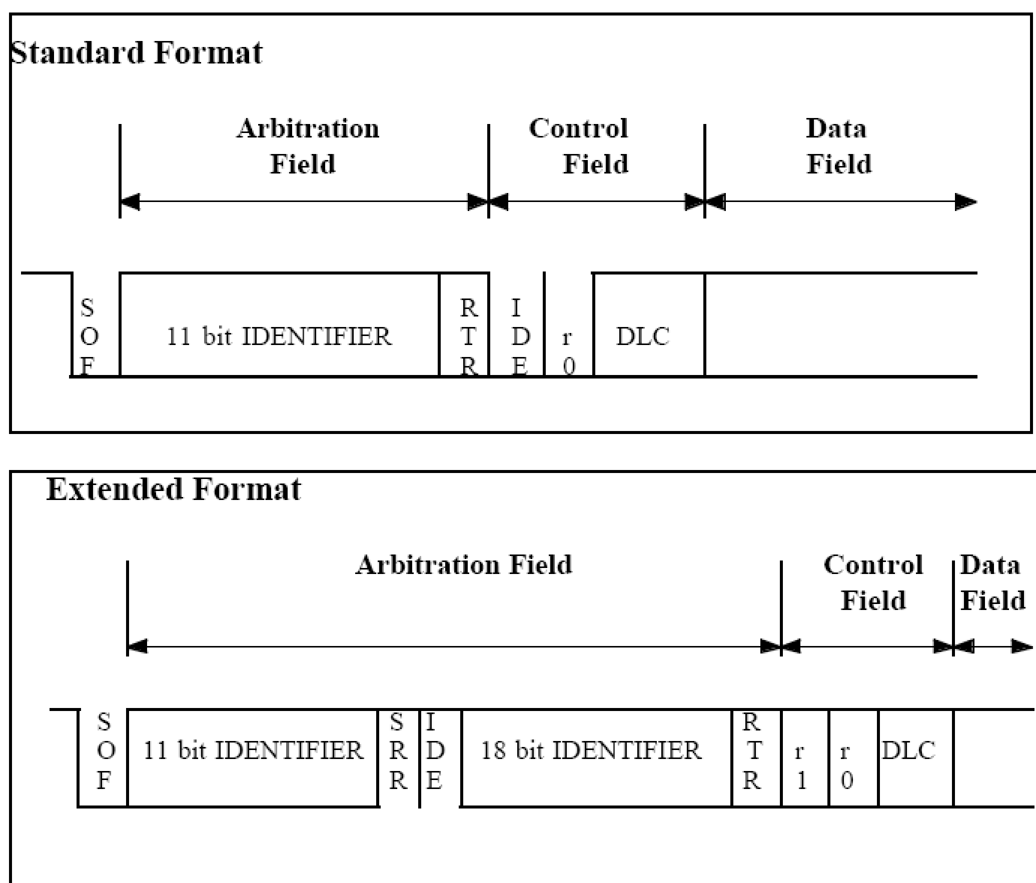


Figura 28: Trama estándar vs trama extendida en CAN

3.8.7 Tramas remotas

Una unidad que funcionando como receptora para un cierto tipo de datos puede iniciar la transmisión de una trama remota cuando quiera solicitar esa información a la unidad que la posea.

Son como las tramas de datos pero tienen el *bit* RTR recesivo y no tienen campo de datos, independientemente del contenido del campo de longitud de datos.

3.9. Bus CAN y CANopen

El bus de campo CAN sólo define las capas física y de enlace por lo que es necesario definir cómo se asignan y utilizan los identificadores y datos de los mensajes CAN. Para ello se definió el protocolo CANopen, que está basado en CAN, e implementa la capa de aplicación. Actualmente está ampliamente extendido, y ha sido adoptado como un estándar internacional.

La construcción de sistemas basados en CAN que garanticen la interoperatividad entre dispositivos de diferentes fabricantes requiere una capa de aplicación y unos perfiles que estandaricen la comunicación en el sistema, la funcionalidad de los dispositivos y la administración del sistema:

- Capa de aplicación (*application layer*). Proporciona un conjunto de servicios y protocolos para los dispositivos de la red.
- Perfil de comunicación (*communication profile*). Define cómo configurar los dispositivos y los datos, y la forma de intercambiarlos entre ellos.
- Perfiles de dispositivos (*device profiles*). Añade funcionalidad específica a los dispositivos.

La relación entre el modelo OSI y los estándares CAN y CANopen la podemos ver en la siguiente figura:

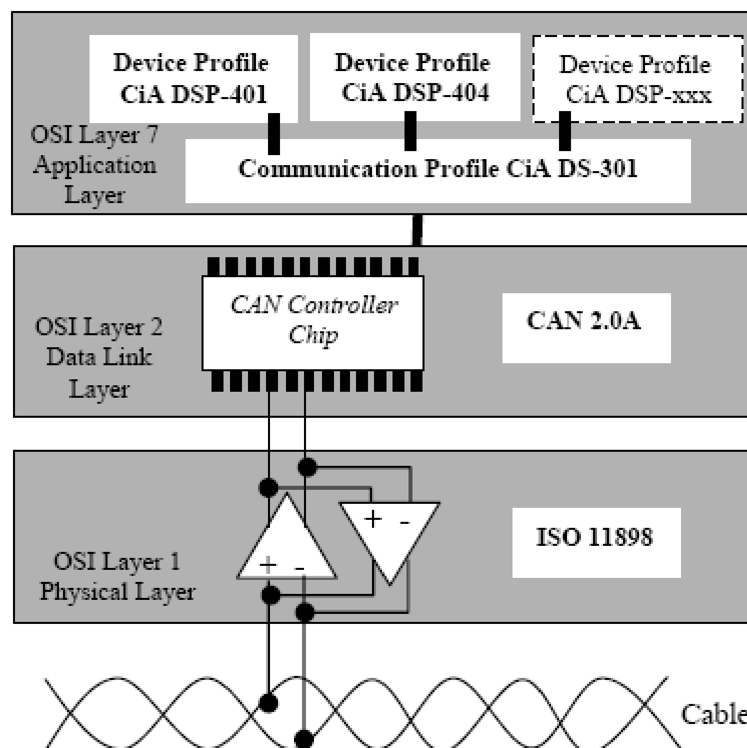


Figura 29: Visión esquemática de los estándares CAN y CANopen en el modelo OSI

A continuación se describirá la capa de aplicación CAL para las redes CAN, en la que se basa el protocolo CANopen. Después se explicará dicho protocolo y los perfiles que define.

3.9.1 CAL (CAN Application Layer)

Fue una de las primeras especificaciones producidas por CiA (CAN in Automation) [30], basada en un protocolo existente desarrollado originalmente por Philips Medical Systems. Ofrece un ambiente orientado a objetos para el desarrollo de aplicaciones distribuidas de cualquier tipo, basadas en CAN.

Esta capa está compuesta por los siguientes cuatro servicios:

- CMS (*CAN-based Message Specification*): ofrece objetos de tipo variable, evento y dominio para diseñar y especificar cómo se accede a la funcionalidad de un dispositivo través de su interfaz CAN.
- NMT (*Network ManagemenT*): proporciona servicios para la gestión de la red. Realiza las tareas de inicializar, arrancar, parar o detectar fallos en los nodos. Se basa en el concepto de maestro-esclavo, habiendo sólo un NMT maestro en la red.
- DBT (*DistriBuTor*): se encarga de asignar de forma dinámica los identificadores CAN (de 11 bits), también llamados COB-ID (*Communication Object Identifier*). También se basa en el modelo maestro-esclavo, existiendo sólo un DBT maestro.
- LMT (*Layer ManagemenT*): permite cambiar ciertos parámetros de las capas como por ejemplo el identificador de un nodo (Node-ID) o la velocidad del bus CAN.

CMS define 8 niveles de prioridad en sus mensajes, cada uno con 220 COB-IDs, ocupando desde el 1 al 1760. Los identificadores restantes (0, 1761-2031) se reservan para NMT, DBT y LMT, como muestra la Tabla 3: Distribución de los COB-IDs en CAL. En una red CAN el mensaje con el COB-ID más pequeño es el de mayor prioridad.

Este estándar asume CAN2.0A (*CAN Standard Message Frame*) con identificadores de 11 bits. Sin embargo se puede utilizar CAN2.0B (*CAN Extended Message Frame*) con identificadores de 29 bits. En ese caso se mapean los 11 bits definidos anteriormente con los 11 bits más significativos del identificador de tramas extendidas. De esta manera los rangos de la tabla quedan bastante más amplios.

COB-ID	Utilización	Cantidad
0	NMT Start/Stop	1
1-220	Objetos CMS Prioridad 0	220
221-440	Objetos CMS Prioridad 1	220
441-660	Objetos CMS Prioridad 2	220
661-880	Objetos CMS Prioridad 3	220
881-1100	Objetos CMS Prioridad 4	220
1101-1320	Objetos CMS Prioridad 5	220
1321-1540	Objetos CMS Prioridad 6	220
1541-1760	Objetos CMS Prioridad 7	220
1761-2015	Monitoreo dispositivos NMT	255
2016-2031	Servicios de NMT, LMT y DBT	16

Tabla 3: Distribución de los COB-IDs en CAL

3.10. CANopen

CAL proporciona todos los servicios de gestión de red y mensajes del protocolo pero no define los contenidos de los objetos CMS ni los tipos de objetos. Es decir, define cómo pero no qué. Aquí es donde CANopen entra en juego.

CANopen [32] está por encima de CAL y utiliza un subconjunto de sus servicios y protocolos de comunicación. Proporciona una implementación de un sistema de control distribuido utilizando los servicios y protocolos de CAL.

El concepto central de CANopen es el diccionario de objetos (OD, *Device Object Dictionary*). Es un concepto utilizado por otros buses de campo. No forma parte de CAL.

3.10.1 Diccionario de objetos de CANopen

Es un grupo ordenado de objetos. Describe completamente y de forma estandarizada la funcionalidad de cada dispositivo y permite su configuración mediante mensajes (SDO) a través del propio bus.

Cada objeto se direcciona utilizando un índice de 16 bits. Para permitir el acceso a elementos individuales de las estructuras de datos también existe un subíndice de 8 bits. En la siguiente tabla podemos ver la estructura general del diccionario:

Indice	Objeto	Descripción
0x0000	No usado	
0x0001 - 0x001F	Tipos de dato estáticos	Contiene definiciones de tipos de dato estándar como boolean, enteros, string, punto flotante, etc. Se incluyen como referencia, no pueden ser leídas ni escritas.
0x0020 - 0x003F	Tipos de dato complejos	Contiene definiciones de estructuras predefinidas, compuestas de tipos estáticos, comunes a todos los dispositivos.
0x0040 - 0x005F	Tipos de dato específicos del fabricante	Contiene definiciones de estructuras predefinidas, compuestas de tipos estáticos, específicas de un dispositivo en particular.
0x0060 - 0x007F	Tipos de dato estáticos específicos del perfil del dispositivo	Definiciones de tipos de dato básicos específicos para el perfil del dispositivo.
0x0080 - 0x009F	Tipos de dato complejos específicos del perfil del dispositivo	Definiciones de estructuras específicas para el perfil del dispositivo.
0x00A0 - 0x0FFF	Reservado	
0x1000 - 0x1FFF	Rango para el perfil de comunicaciones	Contiene parámetros de configuración del bus CAN. Estas entradas del diccionario son comunes a todos los dispositivos.
0x2000 - 0x5FFF	Rango para el perfil específico del fabricante	Contiene las extensiones al perfil estándar, realizadas por el fabricante.
0x6000 - 0x9FFF	Rango para perfiles de dispositivo estandarizados	Contiene todos los objetos de datos comunes a un tipo de perfil que pueden ser leídos o escritos desde la red. Algunas de las entradas son obligatorias (funcionalidad requerida) mientras que otras son opcionales (funcionalidad opcional).
0xA000 - 0xFFFF	Reservado	

Tabla 4: Estructura de un diccionario de objetos estándar en CANopen

El rango relevante de objetos va desde el índice 1000 al 9FFF. Para cada nodo de la red existe un OD, diccionario de objetos, que contiene todo los parámetros que describen el dispositivo y su comportamiento en la red.

En CANopen hay documentos que describen perfiles. Hay un perfil de comunicaciones (*communication profile*) donde están descritos todos los parámetros relacionados con las comunicaciones. Además hay varios perfiles de dispositivos (*device profiles*) donde se definen los objetos de un dispositivo en particular.

Un perfil define para cada objeto del diccionario su función, nombre, índice, subíndice, tipos de datos, si es obligatorio u opcional, si es de “sólo lectura”, “sólo escritura” o “lectura-escritura”, etc.

3.10.2 Identificadores de mensaje

CANopen define la distribución de los identificadores de mensaje (*Predefined Connection Set*) de manera que hay un mensaje de emergencia por nodo, mensajes de sincronización y *time stamp*, un SDO (ocupando dos identificadores), mensajes NMT y cuatro PDOs de transmisión y cuatro de recepción por dispositivo.

El identificador de 11 bits se divide en dos partes:

- 4 bits para el código de función
- 7 bits para el identificador de nodo (Node-ID)

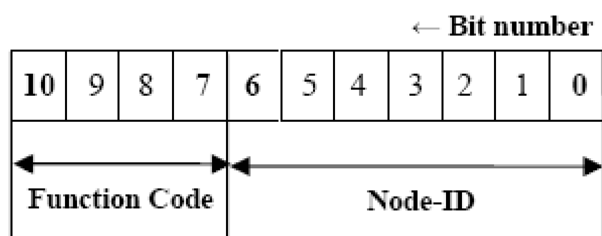


Figura 30: Estructura del identificador de mensajes CAN

La distribución de los identificadores se corresponde con una estructura del tipo maestro-esclavo. El maestro que conoce los Node-IDs de todos los esclavos conectados (máximo 127). Dos esclavos no pueden comunicarse entre sí porque no conocen sus identificadores.

En las siguientes tablas podemos ver la distribución general de los identificadores:

Broadcast objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID**	**Communication parameters at OD index**
NMT Module Control	0000	000h	–
SYNC	0001	080h	1005h, 1006h, 1007h
TIME STAMP	0010	100h	1012h, 1013h

Peer-to-Peer objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID ***	**Communication parameters at OD index**
EMERGENCY	0001	081h - 0FFh	1024h, 1015h
PDO 1 (transmit)	0011	181h - 1FFh	1800h
PDO 1 (receive)	0100	201h - 27Fh	1400h
PDO 2 (transmit)	0101	281h - 2FFh	1801h
PDO 2 (receive)	0110	301h - 37Fh	1401h
PDO 3 (transmit)	0111	381h - 3FFh	1802h
PDO 3 (receive)	1000	401h - 47Fh	1402h
PDO 4 (transmit)	1001	481h - 4FFh	1803h
PDO 4 (receive)	1010	501h - 57Fh	1403h
SDO (transmit/server)	1011	581h - 5FFh	1200h
SDO (receive/client)	1100	601h - 67Fh	1200h
NMT Error Control	1110	701h - 77Fh	1016h, 1017h

Tabla 5: Asignación de los identificadores CAN en CANopen

3.10.3 Modelo de comunicaciones

El modelo de comunicaciones de CANopen define cuatro tipos de mensajes (objetos de comunicación):

- Objetos administrativos: son mensajes administrativos que permiten la configuración de las distintas capas de la red así como la inicialización, configuración y supervisión de la misma. Se basa en los servicios NMT, LMS (LSS) y DBT de la capa CAL.
- *Service Data Objects* (SDO): objetos o mensajes de servicio utilizados para leer y escribir cualquiera de las entradas del diccionario de objetos de un dispositivo. Corresponden a mensajes CAN de baja prioridad.
- *Process Data Objects* (PDO): objetos o mensajes de proceso utilizados para el intercambio de datos de proceso, es decir, datos de tiempo real. Por este motivo, típicamente corresponden a mensajes CAN de alta prioridad.
- Mensajes predefinidos: de sincronización, de emergencia y *time stamp*. Permiten la sincronización de los dispositivos (objetos SYNC) y generar notificaciones de emergencia en forma opcional.

CANopen soporta los modelos de comunicación punto-a-punto, maestro-esclavo y productor-consumidor en sus variantes *push* y *pull*. En el modelo *push* los productores colocan los eventos en el canal de eventos y éste se los envía a los consumidores. En el *pull* el flujo de eventos ocurre en el sentido contrario, es decir, los consumidores solicitan eventos al canal de eventos y éste los solicita a los productores.

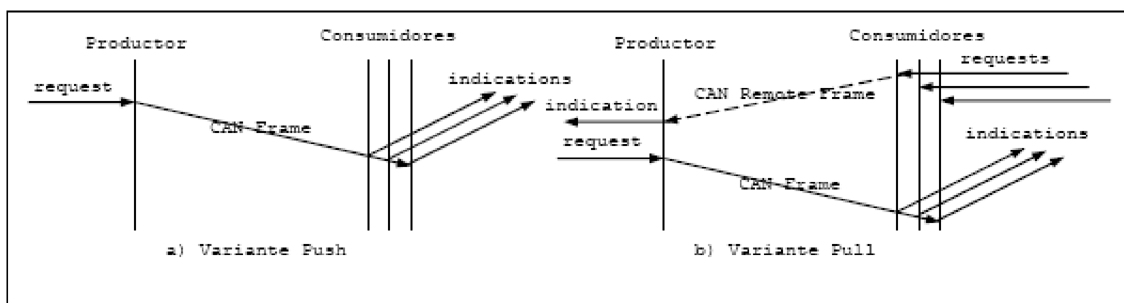


Figura 31: Modelo de comunicación productor-consumidor en CANopen

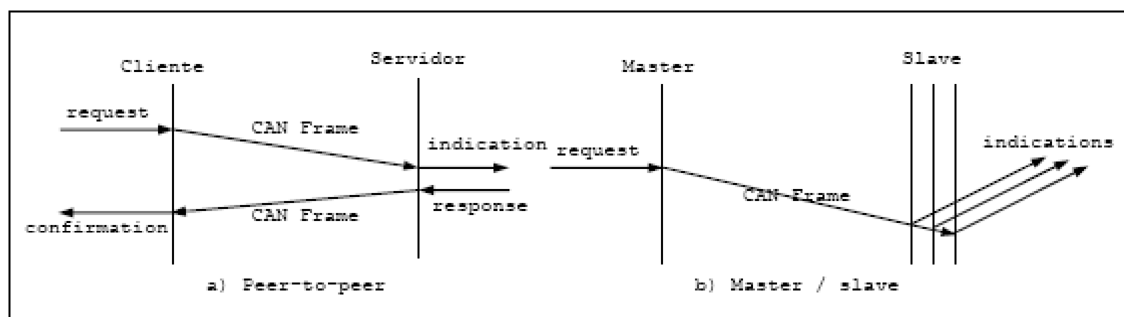


Figura 32: Modelos de comunicación punto-a-punto y maestro-esclavo en CANopen

3.10.4 Service Data Objects (SDO)

Normalmente este tipo de objetos es usado para configuración de dispositivos y transferencia de grandes cantidades de datos no relevantes en forma directa para el control del proceso. En comparación con los PDOs, son mensajes de baja prioridad.

Los objetos de servicio o SDOs permiten implementar los modelos de comunicación cliente-servidor o punto-a-punto, para acceder a los diccionarios de objetos de los dispositivos. Para un cierto SDO, el dispositivo cuyo diccionario está siendo accedido es el servidor mientras que el otro dispositivo, el que inicia la actividad, es el cliente.

Este tipo de objetos ofrece transferencia de datos sin conexión y con confirmación. Por este motivo, cada SDO involucra el intercambio de dos tramas CAN con diferentes identificadores.

Un SDO es representado en CMS como un objeto de tipo *Multiplexed Domain*. Se definen una serie de protocolos petición-respuesta que se pueden aplicar a los SDOs para su transferencia:

- *Initiate Domain Download*
- *Initiate Domain Upload*
- *Download Domain Segment*
- *Upload Domain Segment*
- *Abort Domain Transfer*

Download significa escribir en el diccionario de objetos y *upload* leer del él.

Al ser *Multiplexed Domains* desde el punto de vista de CMS, los SDOs pueden transferir datos de cualquier longitud. Sin embargo CANopen define dos tipos de transferencia para los SDO basándose en el tamaño de los datos a transferir: transferencia expédita, para datos de longitud menor o igual a 4 *bytes*, y transferencia en segmentos, para datos de longitud mayor de 4 *bytes*.

Los mensajes tanto del cliente como del servidor siempre tienen una longitud de 8 *bytes* aunque no todos contengan información significativa.

Es importante tener en cuenta que en CANopen los parámetros de más de un *byte* se envían siempre en la forma *little endian*, es decir, primero el *byte* menos significativo (LSB).

Transferencia expédita

Usada para transmitir mensajes con longitud de datos menor o igual a 4 *bytes*, para lo cual se usan los protocolos *Initiate Domain Download* o *Initiate Domain Upload*.

No se aplica fragmentación, se envía un único mensaje CAN y se recibe la confirmación del servidor.

Un SDO que se transmite del cliente al servidor bajo este protocolo tiene el siguiente formato:

- *Command Specifier* (1 *byte*): contiene información sobre si es subida o descarga de datos, petición o respuesta, transferencia expédita o en segmentos, tamaño de los datos y el *toggle* bit:
  - *Client Command Specifier* (3 bits): 001 para *download* y 010 para *upload*.
  - Ignorado (1 bit): se pone a 0.

  - Número de *bytes* / Ignorado (2 bits): número de *bytes* que no contienen datos. Es válido si los siguientes dos bits son 1. Si no, vale 0. En *upload* se ignora.
  - *Transfer Expedited* / Ignorado (1 bit): indica si se trata de una transferencia expédita (a 1) o una transferencia en segmentos (a 0). Tiene que estar siempre a 1 para este tipo de transferencia. En *upload* se ignora.
  - *Block Size Indicated* / Ignorado (1 bit): si está a 1 es que el tamaño de los datos está especificado, si no, no. En *upload* se ignora.
- *Index* (2 *bytes*): índice de la entrada del diccionario de objetos del servidor que el cliente desea acceder mediante el SDO actual. Este campo y el siguiente forman el multiplexor del dominio.
- *Subindex* (1 *byte*): subíndice de la entrada del diccionario de objetos del servidor que el cliente desea acceder. Sólo tiene sentido si la entrada es de un tipo complejo. Si se trata de una entrada con tipo estático, debe ser 0.
- Datos / Ignorado (4 *bytes*): datos que se desean enviar al servidor (el valor de una entrada en el diccionario si se está escribiendo). Si es *upload* se ignora.

El servidor, al recibir el mensaje anterior, y si ha accedido con éxito al diccionario, contesta con un mensaje con el siguiente formato:

- *Command Specifier* (1 *byte*):
  - *Server Command Specifier* (3 bits): 011 para *download* y 010 para *upload*.
  - Ignorado (1 bit): se pone a 0.
  - Ignorado / Número de *bytes* (2 bits): si es un *download* se ignora.
  - Ignorado / *Transfer Expedited* (1 bit): si es un *download* se ignora.
  - Ignorado / *Block Size Indicated* (1 bit): si es un *download* se ignora.
- Ignorado / Datos (4 *bytes*): si es un *download* (escritura) se ignora ya que es una confirmación. Si es una lectura o *upload*, contiene el valor leído del diccionario de objetos del servidor.

Transferencia en segmentos

Usada para transmitir mensajes con longitud de datos mayor de 4 *bytes*. Se aplica fragmentación en segmentos partiendo los datos en múltiples mensajes CAN. El cliente espera confirmación del servidor por cada segmento.

Para el primer mensaje tanto del cliente como del servidor, se usan los protocolos *Initiate Domain Download* o *Initiate Domain Upload* según corresponda, con el bit *Transfer Expedited* a 0 (para transferencia en segmentos). Si ese bit está a 0 y el siguiente (*Block Size Indicated*) está a 1, significa que el campo de datos (de 4 *bytes*) contiene el número de *bytes* que se van a transmitir (al ser *little endian* el *byte* 4 del mensaje contiene el LSB y el 7 el MSB).

Para los mensajes siguientes se usan los protocolos *Download Domain Segment* o *Upload Domain Segment*, según se quiera leer o escribir una entrada en el diccionario. Los mensajes CAN bajo estos dos protocolos tienen el siguiente formato:

- *Command Specifier* (1 *byte*):
    - *Client Command Specifier* (3 bits): 000 para *download* y 011 para *upload*.
    - *Toggle Bit* (1 bit): es un bit que vale 0 o 1 de forma alternada en segmentos consecutivos. La primera vez vale 0. Teniendo en cuenta que el mecanismo de intercambio sólo permite un mensaje pendiente de confirmación, un único bit es suficiente para esto. El servidor se limita a hacer un eco del valor recibido.
    - Número de *bytes* / Ignorado (3 bits): indica el número de *bytes* que no contienen datos, o cero si no especifica el tamaño. Se ignora en el *upload*.
    - *Last Segment Indication* / Ignorado (1 bit): vale 1 si se trata del último segmento del SDO que se está transmitiendo y 0 si hay más segmentos. Para *upload* se ignora.
- Datos / Ignorado (7 *bytes*): datos que se desean enviar al servidor. Son los *bytes* que no caben en el mensaje CAN inicial. Si es *upload* se ignora.

El servidor, al recibir el mensaje anterior, contesta con un mensaje con el formato:

- *Command Specifier* (1 *byte*):
    - *Client Command Specifier* (3 bits): 001 para *download* y 000 para *upload*.
    - *Toggle Bit* (1 bit): igual que antes.
    - Ignorado / Número de *bytes* (3 bits): se ignora en el *download*.
    - Ignorado / *Last Segment Indication* (1 bit): se ignora en el *download*.
- Ignorado / Datos (7 *bytes*): si es un *download* (escritura) se ignora porque es una confirmación. Si es una lectura o *upload*, contiene el valor leído del diccionario de objetos del servidor.

Abort Domain Transfer

Uno de los protocolos antes mencionados y que aún no se ha explicado es el *Abort Domain Transfer*. Existe la posibilidad que al acceder a las entradas del diccionario de objetos del servidor, se produzca un error. Este protocolo es usado en esos casos para notificar tanto a clientes como a servidores. El formato de los mensajes de este protocolo es el siguiente:

- *Command Specifier* (1 *byte*):
    - *Command Specifier* (3 bits): 100 para *Abort Domain Transfer*.
    - Ignorado (5 bits): se ignoran.
- *Index* (2 *bytes*): índice de la entrada del diccionario de objetos del servidor que causó el error que se está notificando. Este campo y el siguiente forman el multiplexor del dominio.
- *Subindex* (1 *byte*): subíndice de la entrada del diccionario de objetos del servidor que causó el error que se está notificando. Sólo tiene sentido si la entrada es de un tipo complejo. Si se trata de una entrada con tipo estático, debe ser 0.
- Código de error (4 *bytes*): código que identifica el error. En la siguiente tabla podemos ver sus posibles valores:

Abort Code	Description
0503 0000	Toggle bit not alternated
0504 0000	SDO protocol timed out
0504 0001	Client/Server command specifier not valid or unknown
0504 0002	Invalid block size (Block Transfer mode only)
0504 0003	Invalid sequence number (Block Transfer mode only)
0503 0004	CRC error (Block Transfer mode only)
0503 0005	Out of memory
0601 0000	Unsupported access to an object
0601 0001	Attempt to read a write-only object
0601 0002	Attempt to write a read-only object
0602 0000	Object does not exist in the Object Dictionary
0604 0041	Object can not be mapped to the PDO
0604 0042	The number and length of the objects to be mapped would exceed PDO length
0604 0043	General parameter incompatibility reason
0604 0047	General internal incompatibility in the device
0606 0000	Object access failed due to a hardware error
0606 0010	Data type does not match, lengh of service parameter does not match
0606 0012	Data type does not match, lengh of service parameter is too high
0606 0013	Data type does not match, lengh of service parameter is too low
0609 0011	Sub-index does not exist
0609 0030	Value range of parameter exceeded (only for write access)
0609 0031	Value of parameter written too high
0609 0032	Value of parameter written too low
0609 0036	Maximum value is less than minimum value
0800 0000	General error
0800 0020	Data can not be transferred or stored to the application
0800 0021	Data can not be transferred or stored to the application because of local control
0800 0022	Data can not be transferred or stored to the application because of the present device state
0800 0023	Object Dictionary dynamic generation fails or no Object Dictionary is present (e.g. OD is generated from file and generation fails because of a file error)

Tabla 6: Códigos de error para SDO *Abort Domain Transfer*

Los SDOs requieren ser definidos en el diccionario de objetos mediante una estructura que contiene parámetros relacionados con la transmisión de los mismos. La estructura para el primer SDO para servidores tiene un índice de 0x1200 mientras que el primer SDO para clientes, se ubica en la entrada 0x1280. En total, en una red CANopen, se pueden definir hasta 128 SDOs para clientes y 128 SDOs para servidores.

3.10.5 Process Data Objects (PDO)

Este tipo de objetos permite intercambiar datos del proceso en tiempo real. Implementa el modelo de comunicaciones productor-consumidor. Los datos se transmiten desde un productor a varios consumidores.

Ofrece un servicio de transferencia de datos sin conexión y sin confirmación. No se aplica un protocolo de fragmentación y reensamble de los objetos. Los PDOs están pensados para tráfico de tiempo real de alta prioridad, por lo que es conveniente evitar la sobrecarga que produciría agregar un protocolo de fragmentación y confirmación como el que se usa en los SDOs.

Los mensajes PDO de un nodo o dispositivo pueden dividirse en dos categorías. Los tPDO son aquello mensajes con información del proceso que el nodo transmite (por ejemplo la lectura de un sensor). Por otro lado, los rPDO son los mensajes con información del proceso que el

nodo escucha (por ejemplo un nodo que controle la apertura de una bomba escuchará el bus en busca de órdenes).

El contenido de un PDO está definido tan sólo por su identificador. Tanto el emisor como el receptor deben conocerlo para poder interpretar su estructura interna. Cada PDO se describe mediante dos objetos del diccionario:

- PDO *Communication Parameter*: contiene el COB-ID que utiliza el PDO, el tipo de transmisión, tiempo de inhibición y temporizador.
- PDO *Mapping Parameter*: contiene una lista de objetos del OD contenidos en la PDO, incluyendo su tamaño en bits.

CANopen define varios mecanismos de comunicación para la transmisión de PDOs:

- Transmisión asíncrona:
  - Eventos: la transmisión de un mensaje es causada por la ocurrencia de un evento específico definido en el perfil del dispositivo.
  - Temporizador: existe un temporizador que cada cierto tiempo cause la transmisión.
  - Solicitud remota: la transmisión asincrónica de mensajes PDO puede comenzar al recibir una solicitud remota (trama RTR) enviada por otro dispositivo.
- Transmisión sincrónica: la transmisión sincrónica de mensajes PDO es disparada por la expiración de un período de transmisión, sincronizado mediante la recepción de objetos SYNC. Es decir, cada vez que llega un mensaje SYNC, se abre una ventana de transmisión sincrónica. Los PDOs sincrónicos deben ser enviados dentro de esa ventana. Se distinguen dos modos dentro de este tipo de transmisión:
  - Modo cíclico: son mensajes que se transmiten dentro de la ventana abierta por el objeto SYNC. No se transmiten en todas las ventanas sino con cierta periodicidad, especificada por el campo *Transmission Type* del *Communication Parameter* correspondiente.
  - Modo acíclico: son mensajes que se transmiten a partir de un evento de la aplicación. Se transmiten dentro de la ventana pero no de forma periódica.

En la siguiente tabla podemos ver los distintos modos de transmisión de PDOs, definidos por el *Transmission Type* (entero de 8 bits) del *Communication Parameter*:

Trans-Mission Type	Condition to trigger PDO (B=both needed, O=one or both)			PDO Transmission
	SYNC*	RTR*	Event*	
0	B	-	B	Sync, acyclic
1-240	O	-	-	Sync, cyclic
241-251	-	-	-	*reserved*
252	B	B	-	Sync, after RTR
253	-	O	-	Async, after RTR
254	-	O	O	Async, manufacturer specific event
255	-	O	O	Async, device profile specific event

*SYNC = objeto SYNC recibido
*RTR = recibida trama RTR
*Event = cambio de valor de un dato, temporizador…

Tabla 7: Modos de transmisión de PDOs enCANopen

Un PDO puede tener asignado un tiempo de inhibición que define el tiempo mínimo que debe pasar entre dos transmisiones consecutivas del mismo PDO. Forma parte del *Communication Parameter*. Está definido como un entero de 16 bits en unidades de 100 microsegundos.

3.10.6 Mensajes adicionales

Mensaje deTime Stamp

Este tipo de objetos representan una cantidad absoluta de tiempo en milisegundos desde el 1 de Enero de 1984. Proporciona a los dispositivos un tiempo de referencia común.La etiqueta temporal o *time-stamp* se implementa como una secuencia de 48 bits.

Mensaje de sincronización (SYNC)

En una red CANopen, hay un dispositivo que es el productor de objetos SYNC y una serie de dispositivos consumidores de objetos SYNC. Cuando los consumidores reciben el mensaje del productor, abren su ventana de sincronismo y pueden ejecutar sus tareas sincrónicas.

Este mecanismo permite coherencia temporal y coordinación entre los dispositivos. Por ejemplo, un conjunto de sensores pueden leer las variables del proceso controlado en forma coordinada y obtener así una imagen consistente del mismo.

El COB-ID usado por este objeto de comunicación puede encontrarse en la entrada 0x1005 del diccionario. Para garantizar el acceso de estos objetos al bus, debería asignárseles un COB-ID bajo. El conjunto predefinido de conexiones de CANopen sugiere usar un valor de 128. El campo de datos del mensaje CAN de este objeto se envía vacío.

El comportamiento de estos mensajes es determinado por dos parámetros:

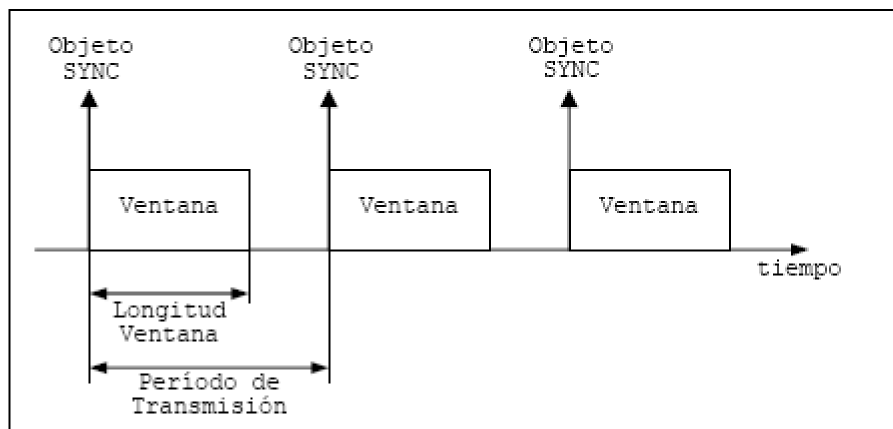


Figura 33: Parámetros de un objeto SYNC en CANopen

La longitud de la ventana o Synchronous Window Length puede ubicarse en la entrada 0x1007 del diccionario. El período de transmisión se encuentra en la posición 0x1006.

Mensaje de emergencia

Estos mensajes se envían cuando ocurre un error interno en un dispositivo. Se transmiten al resto de dispositivos con la mayor prioridad. Pueden usarse como interrupciones o notificaciones de alertas.

Un mensaje de emergencia tiene 8 *bytes*. Su estructura es la siguiente:

COB-ID	Byte 0-1	Byte 2	Byte 3-7
0x080 + Node_ID	**Emergency Error Code**	**Error Register** (Object 0x1001)	Manufac-turer specific error field

Figura 34: Estructura de un mensaje de emergencia en CANopen

- *Emergency Error Code* (2 *bytes*): código del error que causó la generación del mensaje de emergencia. Se trata de fallos internos de los dispositivos por lo cual, los errores se relacionan con fallos de tensión, de corriente, del software del dispositivo, del adaptador del bus CAN, etc. La siguiente tabla nos muestra los códigos correspondientes en hexadecimal:

Emergency Error Code	Meaning
00xx	Error Reset or No Error
10xx	Generic Error
20xx	Current
21xx	current, device input side
22xx	current, inside the device
23xx	current, device output side
30xx	Voltage
31xx	mains voltage
32xx	voltage inside the device
33xx	output voltage
40xx	Temperature
41xx	ambient temperature
42xx	device temperature
50xx	Device hardware
60xx	Device software
61xx	internal software
62xx	user software
63xx	data set
70xx	Additional modules
80xx	Monitoring
81xx	Communication
8110	CAN overrun
8120	Error Passive
8130	Life Guard Error or Heartbeat Error
8140	Recovered from Bus-Off
82xx	Protocol Error
8210	PDO not processed due to length error
8220	Length exceeded
90xx	External error
F0xx	Additional functions
FFxx	Device specific

‘xx’ es la parte dependiente del perfil del dispositivo

Tabla 8: Códigos de error para los mensajes de emergencia de CANopen

- *Error Register* (1 *byte*): entrada con índice 0x1001 del diccionario de objetos. Cada bit de este registro indica una condición de error distinta cuando está a ‘1’. El significado de cada bit es:

Bit	Significado
0	Error genérico.
1	Problema de corriente.
2	Problema de tensión.
3	Problema de temperatura.
4	Error de comunicaciones.
5	Específico del perfil de dispositivo.
6	Reservado.
7	Específico del fabricante del dispositivo.

Tabla 9: Bits del Error Register de los mensajes de emergencia de CANopen

- *Manufacturer-specific Error Field* (5 *bytes*): este campo puede usarse para información adicional sobre el error. Los datos incluidos y su formato son definidos por el fabricante del dispositivo.

Mensajes de Node/Life Guarding

Se utiliza para saber si un nodo está operativo. La comunicación se basa en el concepto de maestro-esclavo. El NMT maestro monitorea el estado de los nodos. A esto se le llama *node guarding*. Opcionalmente los nodos pueden monitorear el estado del NMT maestro. A esto se le llama *life guarding*. Comienza en el NMT esclavo después de haber recibido el primer mensaje de *node guarding* del NMT maestro.

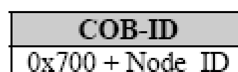
Sirve para detectar errores en las interfaces de red de los dispositivos, pero no fallos en los dispositivos en sí (ya que estos son avisados mediante mensajes de emergencia).

Se puede implementar de dos maneras distintas, pudiendo utilizarse sólo una de ellas a la vez: NMT *Node Guarding* o *Heartbeat*.

NMT *Node Guarding*

El maestro va preguntando a los esclavos si están vivos cada cierto tiempo. Si alguno no contesta en un tiempo determinado significa que está caído y se informa de ello. Si los esclavos también monitorean al maestro deben informar de que el maestro está caído si no reciben mensajes de *Node Guarding* de él durante un determinado intervalo.

El maestro interroga a los esclavos mediante una trama remota (RTR) con la siguiente estructura:



COB-ID
0x700 + Node_ID

Figura 35: Trama RTR que el NMT maestro envía a los NMT esclavos

Los NMT esclavos responden con el siguiente mensaje:

COB-ID	Byte 0
0x700 + Node_ID	bit 7: *toggle*, bit 6-0: *state*

Figura 36: Mensaje que los NMT esclavos envían al NMT maestro

El *toggle bit* (bit 7) es un bit que va alternando su valor en cada mensaje de *Node Guarding*. La primera vez vale ‘0’.

Los bits del 0 al 6 indican el estado del nodo. Su valor lo podemos ver en la siguiente tabla:

Value	State
0	Initialising
1	Disconnected *
2	Connecting *
3	Preparing *
4	Stopped
5	Operational
127	Pre-operational

Los estados marcados con * son para si realiza un *boot-up* extendido

Tabla 10: Valor del campo *state* en un mensaje de NMT *Node Guarding*

Heartbeat

Cada nodo manda un mensaje de *Hearbeat* cada cierto tiempo para informar de que está operativo. En este caso el mensaje de *Boot-up* se considera que es el primer mensaje de *Heartbeat*. Si el NMT maestro deja de recibir estos mensajes durante un tiempo determinado significará que el nodo está caído.

Tienen la siguiente estructura:

COB-ID	Byte 0
0x700 + Node_ID	*state*

Figura 37: Estructura de un mensaje de *Heartbeat*

state	Meaning
0	Boot-up
4	Stopped
5	Operational
127	Pre-operational

Tabla 11: Valor del campo *state* en un mensaje de *Heartbeat*

3.10.7 Gestión de la red (NMT)

Aparte de los mensajes predefinidos, CANopen incluye una serie de mensajes para la administración y monitoreo de los dispositivos en la red. Estos están implementados en la capa CAL y reciben el nombre de servicios de gestión de red (*Network ManagemenT*, NMT). Se trabaja con un modelo de comunicaciones maestro-esclavo en el cual un dispositivo es el NMT maestro y el resto los NMT esclavos.

Un dispositivo NMT esclavo puede encontrarse en alguno de los siguientes estados:

- ***Initialising***: al encender el dispositivo se pasa directamente a este estado. Después de realizar las labores de inicialización el nodo transmite el mensaje de *Boot-up* y pasa al estado *Pre-Operational*.

COB-ID	Byte 0
0x700 + Node_ID	0

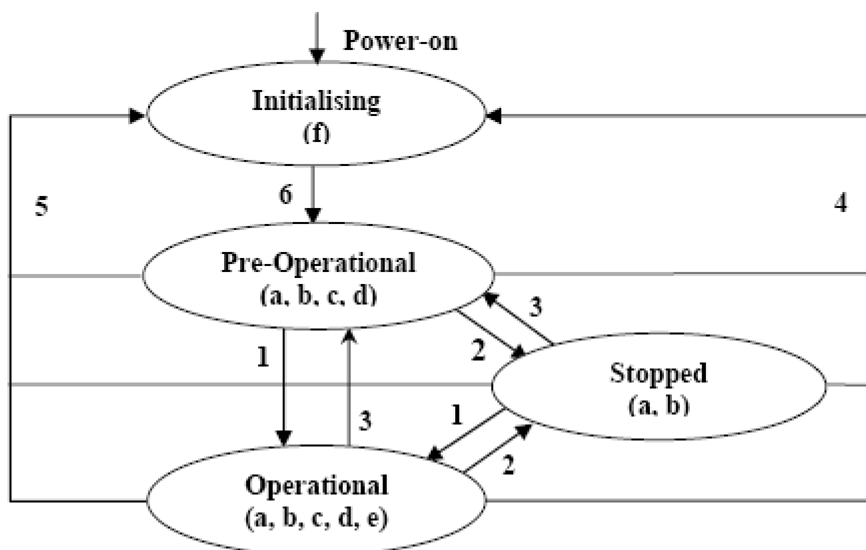
Figura 38: Estructura del mensaje de *Boot-up*

  Para permitir un reseteo parcial del nodo se subdivide en tres subestados:

  - *Reset-Application*: los parámetros específicos del fabricante y del perfil del dispositivo son fijados a su valor inicial. A continuación el nodo pasa al estado *Reset-Communication*.
  - *Reset-Communication*: los parámetros del perfil de comunicaciones son fijados a su valor inicial. A continuación el nodo pasa al estado *Initialising*.

- ***Pre-operational***: en este estado el dispositivo puede ser configurado mediante SDOs. Puede enviar y recibir SDOs, mensajes de emergencia, de sincronización, *time stamp* y mensajes NMT.
- ***Operational***: el dispositivo ya ha sido configurado y funciona normalmente. Todos los objetos de comunicación están activos así también puede enviar y recibir PDOs.
- ***Stopped***: todos los objetos de comunicación dejan de estar activos. No se pueden enviar ni recibir PDOs ni SDOs, sólo mensajes de NMT.

A continuación podemos ver en detalle el diagrama de los posibles estados de un nodo:



Las letras entre paréntesis indican qué objetos de comunicación están permitidos en cada estado:

a. NMT, b. Node Guard, c. SDO, d. Emergency, e. PDO, f. Boot-up

Transiciones entre estados (mensajes NMT):

1: Start_Remote_Node (command specifier 0x01)

2: Stop_Remote_Node (command specifier 0x02)

3: Enter_Pre-Operational_State (command specifier 0x80)

4: Reset_Node (command specifier 0x81)

5: Reset_Communication (command specifier 0x82)

6: Inicialización terminada, entra en Pre-Operational directamente al mandar el mensaje de Boot-up

Figura 39: Diagrama de transición de estados de un nodo en CANopen

Sólo el NMT maestro puede mandar mensajes del módulo de control NMT, pero todos los esclavos deben dan soporte a los servicios del módulo de control NMT.

No hay respuesta para un mensaje NMT. Sólo los envía el maestro y tienen el siguiente formato:

COB-ID	Byte 0	Byte 1
0x000	*CS*	Node-ID

Figura 40: Estructura de un mensaje NMT

Con el Node-ID (identificador de nodo) igual a cero, todos los NMT esclavos son diseccionados (*broadcast*).

El campo CS (*Command Specifier*) puede tener los siguientes valores:

Command Specifier	NMT Service
1	Start Remote Node
2	Stop Remote Node
128	Enter Pre-operational State
129	Reset Node
130	Reset Communication

Tabla 12: Valores del campo CS del mensaje NMT

4. Técnicas y herramientas

4.1. Técnicas metodológicas

En el subsistema de los microcontroladores se ha utilizado la programación estructurada, ya que es la forma de realizar este tipo de programas. Tan sólo se utilizan tres estructuras de control: secuencial, selectiva (o condicional) e iterativa. Con ellas es suficiente, y se crean programas más sencillos y más rápidos.

Para realizar la interfaz gráfica de usuario nos hemos decantado por el paradigma objetual, ya que creemos, se ajusta mejor a nuestras necesidades. La orientación a objetos mejora la calidad del software, acorta los tiempos de desarrollo, aumenta la productividad e incrementa la reutilización. Estas son unas características muy deseables. Además tiene una serie de ventajas frente a la programación estructurada como son [6]:

- ✓ **Uniformidad**. Ya que la representación de los objetos lleva implícitos tanto el análisis como el diseño y la codificación de los mismos.
- ✓ **Comprensión**. Tanto los datos que componen los objetos, como los procedimientos que los manipulan, están agrupados en clases, que se corresponden con las estructuras de información que el programa trata. Esto proporciona una división más natural de los sistemas.
- ✓ **Flexibilidad**. Al tener relacionados los procedimientos que manipulan los datos con los datos a tratar, cualquier cambio que se realice sobre ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan.
- ✓ **Estabilidad**. Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo sobre aquellos que cambian con frecuencia permite aislar las partes del programa que permanecen inalterables en el tiempo. De esta manera se mejora el mantenimiento y la evolución de los sistemas software.
- ✓ **Reutilización**. La noción de objeto permite que programas que traten las mismas estructuras de información reutilicen las definiciones de objetos empleadas en otros programas e incluso los procedimientos que los manipulan. De esta forma, el desarrollo de un programa puede llegar a ser una simple combinación de objetos ya definidos donde éstos están relacionados de una manera particular. La reutilización sistemática del software proporciona las bases para el incremento de la calidad, fiabilidad y a largo plazo reduce los costes del desarrollo y el mantenimiento del software.

4.1.1 UML (Unified Modeling Language)

El Lenguaje Unificado de Modelado (UML) es un lenguaje de modelado visual que se usa para especificar, visualizar, y construir y documentar artefactos de un sistema software. Captura decisiones y conocimiento sobre los sistemas que se deben construir. Se usa para entender, diseñar, hojear, configurar, mantener y controlar la información sobre tales sistemas. Está pensado para usarse con todos los métodos de desarrollo, etapas del ciclo de vida, dominios de aplicación y medios. El lenguaje de modelado pretende unificar la experiencia pasada sobre técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar. UML incluye conceptos semánticos, notación y principios generales. Pretende dar apoyo a la mayoría de los procesos de desarrollo orientados a objetos [4].

Se ha convertido en el estándar de facto de la industria, debido a que ha sido impulsado por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh. Estos autores fueron contratados por la empresa Rational Software Co. para crear una notación unificada en la que basar la construcción de sus herramientas CASE. En el proceso de creación de UML han participado, no obstante, otras empresas de gran peso en la industria como Microsoft, Hewlett-Packard, Oracle o IBM, así como grupos de analistas y desarrolladores.

Esta notación ha sido ampliamente aceptada debido al prestigio de sus creadores y a que incorpora las principales ventajas de cada uno de los métodos particulares en los que se basa (principalmente Booch, OMT y OOSE). UML ha puesto fin a las llamadas “guerras de métodos” que se han mantenido a lo largo de los 90, en las que los principales métodos sacaban nuevas versiones que incorporaban las técnicas de los demás. Con UML se fusiona la notación de estas técnicas para formar una herramienta compartida entre todos los ingenieros software que trabajan en el desarrollo orientado a objetos [48].

4.2. Lenguajes de programación

4.2.1 C

C es un lenguaje de programación creado en 1969 por Ken Thompson y Dennos M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, posee construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del C fue por ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que muchas veces el código puede portarse entre plataformas y/o arquitecturas.

Se ha elegido el lenguaje C para desarrollar los programas de los microcontroladores porque además de ser un lenguaje indicado para la programación a más bajo nivel, ya que permite realizar fácilmente este tipo de operaciones, existen multitud de compiladores que generan el código en ensamblador, lo cual hace mucho más cómoda y fácil la programación.

4.2.2 Java

Java es un lenguaje de programación creado por Sun Microsystems. Está basado en C y C++. Es sencillo, seguro e independiente de la plataforma. Estas tres características le han hecho muy popular entre los programadores [7].

La portabilidad se consigue de la siguiente manera: el código fuente se compila a un lenguaje intermedio (llamado *bytecodes*) cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta y después se interpreta por medio de un programa denominado máquina virtual de Java.

Después de barajar varias opciones, Java (versión 1.5) fue el lenguaje de programación elegido para desarrollar la parte de interfaz gráfica del proyecto, ya que cumple dos requisitos fundamentales: se basa en la metodología orientada objetos y es multiplataforma.

También ha sido necesario el uso del **JFC** y **Swing** para construir la interfaz gráfica. JFC es la abreviatura de *Java Foundation Classes*, que comprende un grupo de características para ayudar a construir interfaces gráficos de usuario (GUIs) [12]:

- AWT (*Abstract Window Toolkit*): APIs que permiten a los programas incorporar un sistema de ventanas, incluyendo APIs para "arrastrar y soltar".
- Java 2D: APIs para permitir incorporar fácilmente gráficos 2D de alta calidad, texto e imágenes en aplicaciones y applets Java.
- Componentes SWING: APIs que extienden a las de AWT y aportan una biblioteca de componentes gráficos más rica y extensa, soportando aspecto y comportamiento conectable. Es decir, le ofrece a cualquier componente Swing una amplia selección de aspectos y comportamientos. Por ejemplo, el mismo programa puede usar el Aspecto y Comportamiento Java o el Aspecto y Comportamiento Windows.
- Accesibilidad: APIs para proporcionar accesibilidad a los usuarios con discapacidades.
- Internacionalización: Todas las tecnologías JFC incluyen soporte para crear aplicaciones que puedan interactuar con usuarios a lo largo de todo el mundo utilizando su propio idioma y convenciones culturales.

Para poder acceder al puerto serie una opción es utilizar la API JavaComm de comunicaciones de Sun. Pero su última versión Java Communications 3.0 API [13], no está disponible para Windows. Por esta razón nos hemos decantado por utilizar la **RXTX** [14]. Es una biblioteca de funciones nativa que proporciona acceso a los puertos serie y paralelo para el JDK. Es compatible con la API Comm de Sun. Está disponible para una amplia gama de plataformas y es gratuita. Nosotros hemos utilizado la RXTX 2.1 para usar sin la API Comm (espacio de nombres *gnu.io*).

4.3. Entorno de desarrollo

Durante el proceso de desarrollo se han utilizado varias herramientas, en función de las características de cada una. A continuación se explican las más interesantes:

4.3.1 SourceBoost IDE

SourceBoost IDE [19] es un entorno de desarrollo integrado para una generación de código ensamblador más rápida. El código fuente puede ser desarrollado en un lenguaje de más alto nivel y compilado utilizando cualquiera de los compiladores de SourceBoost [18] como BoostC, BoostBasic, C2C-plus, C2C o P2C-plus.

Algunas de las principales características que ofrece la herramienta son:

- Manejo de proyectos.
- Sintaxis coloreada, función de auto-completar, mostrar los prototipos de las funciones mediante *tooltips*, etc.
- Depurador incorporado.
- Simulador de señal de reloj para las familias de microcontroladores PIC12, PIC16 y PIC18.

- Permite trabajar con diferentes compiladores (nombrados anteriormente).
- Asistente para la creación de proyectos.

Esta herramienta junto con su compilador BoostC ha sido elegida para el desarrollo del código de los microcontroladores, ya que posee ciertas funciones de ayuda para el trabajo con el bus I2C, y genera código ensamblador para el PIC18F258, que es el modelo que usamos.

Hay una versión limitada con licencia libre, pero el departamento ha adquirido una licencia porque las características de la versión libre no nos eran suficientes, ya que superábamos la cuota de código que permitía.

Para instalarlo hay que descargar el *SourceBoost Package*, que ya incluye el entorno de desarrollo junto con los compiladores. Está disponible en:

http://www.sourceboost.com/CommonDownload.html

Durante el desarrollo se ha utilizado la versión 6.60. El instalador para Windows de la herramienta está en el CD-ROM entregado con el proyecto, dentro de la carpeta “Entorno de desarrollo”. También se adjunta un manual de usuario de la aplicación en el directorio “Documentación de apoyo/SourceBoost IDE y BoostC”.

4.3.2 BoostC

BoostC [20] es un compilador de SourceBoost para el lenguaje C que genera código ensamblador para las familias de microcontroladores PIC18, PIC16 y algunos PIC12. Entre otras cosas permite trabajar con tipos de datos con o sin signo, estructuras y punteros.

4.3.3 Eclipse SDK

Eclipse SDK es un polivalente y profesional entorno de desarrollo integrado. Se adapta a las necesidades de cada proyecto a desarrollar debido al gran número de *plug-ins* disponibles para su ampliación y adaptación. Por esta razón permite el desarrollo de aplicaciones de diversas características y en distintos lenguajes.

Posee varias características que hacen que más cómoda la programación como: sintaxis coloreada, función de auto-completar, mostrar los prototipos de las funciones mediante *tooltips*, generación automática de código, ordenación automática del código, ayudas en la depuración, etc.

Es multiplataforma. Para ejecutarlo es necesario tener instalado un JRE (*Java Runtime Environment*) que se puede descargar de [12].

Es un software libre y está disponible en [16]:

http://www.eclipse.org/downloads/

Se eligió este entorno de desarrollo integrado para realizar la parte de la interfaz Java por ser libre, agradable y fácil de usar, además de hacer más cómoda la programación. También por sus grandes capacidades de ampliación y adaptación mediante *plug-ings*.

El instalador está en la carpeta de “Entorno de desarrollo” del CD-ROM entregado junto con el proyecto.

4.3.4 Programadora PICkit 2 y adaptador PGM2KIT

El PICkit 2 [25] es una programadora de microcontroladores de bajo coste para PICs de Microchip, entre ellos el que nosotros usamos (PIC18F258). Está diseñado para ayudar al desarrollador a cargar los programas en los microcontroladores de una manera fácil y rápida.

La programadora (Figura 41) se conecta al PC mediante USB y a la placa del PIC en el conector ICSP[2], con el adaptador PGM2KIT [26]. Para cargar el programa en el PIC se ejecuta el software PICkit 2, que se puede descargar en [24] y que funciona bajo Windows.

Figura 41: Programadora PICkit 2

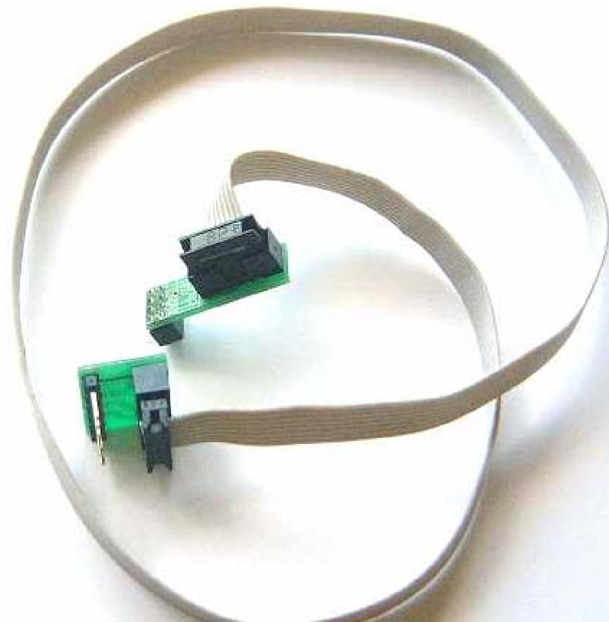


Figura 42: Adaptador PGM2KIT

[2] *In-Circuit Serial Programming*. Programación del PIC en el circuito definitivo a usar, sin necesidad de sacarlo.

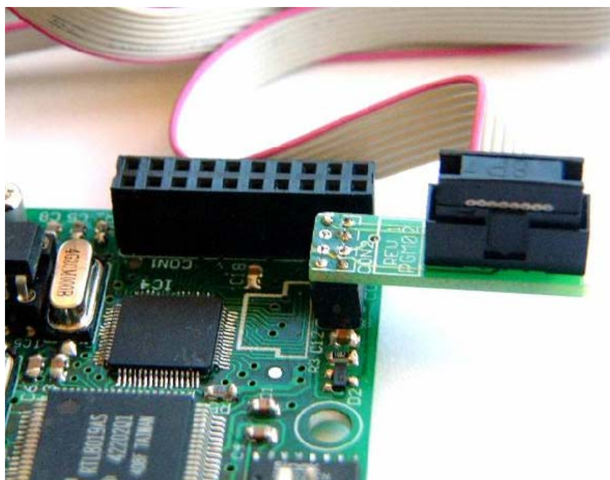


Figura 43: Conexión a la placa mediante un conector ICSP del tipo ICPC1

La programadora y el adaptador PGM2KIT han sido adquiridas recientemente por el departamento. La versión de software utilizada durante el desarrollo fue la 2.10, auque ya está disponible la 2.30. El instalador está en la carpeta de "Entorno de desarrollo" del CD-ROM entregado junto con el proyecto.

4.3.5 232Analyzer

232Analyzer [41] es una herramienta para monitorizar y controlar dispositivos serie (RS232/RS485/RS422/TTL) en el PC. Gracias a él se puede controlar, monitorizar, revisar errores, generar informes, espiar, capturar, visualizar, analizar y probar la actividad en puertos serie. Permite ver todos los datos de comunicación entre dos dispositivos que utilicen estos puertos. Además incluye funciones avanzadas de control de estados, macros, etc.

Algunas de sus características son:

- Monitorización de puertos series.
- Dos modos de operación: monitorización y solución de errores.
- Envía y recibe datos en todos los formatos.
- Envía y recibe señales RTS, DTR, CTS, DSR, CD y RI.
- Del COM 1 al 16.
- Macros programables.
- Calculadora.
- Respuesta automática programable.
- Disponible para Windows.

Se ha utilizado esta herramienta para probar y depurar los programas de los PICs mediante el envío/recepción de datos por el puerto serie.

4.3.6 Doxygen

Doxygen [17] es un sistema de generación de documentación para C++, C, Java, *Objective-C*, *Python*, IDL y algunas extensiones de PHP, C# y D. Es una herramienta libre y funciona en la mayoría de sistemas Unix así como en Windows y Mac OS X.

Extrae los comentarios directamente desde el código fuente, y permite generar la documentación en diferentes formatos de salida como HTML, LaTeX, RTF, *PostScript*, PDF con hipervínculos, HTML comprimido, y páginas *man* para Unix.

Se puede descargar de forma gratuita de [17]:

http://www.doxygen.org

La principal razón para optar por esta herramienta ha sido que nos permitía generar la documentación de todos los subsistemas, ya estuvieran desarrollados en C o en Java. También se ha elegido por la gran variedad de formatos de salida, porque es libre, y por ser multiplataforma.

La documentación de cada subsistema se ha generado e incluido en el CD-ROM en HTML, PDF y LaTeX.

4.3.7 Visual Paradigm for UML Standard Edition 6.0

Es una herramienta CASE que utiliza UML como lenguaje de modelado. Facilita el desarrollo, en las fases de análisis y diseño, de los sistemas software que utilizan una metodología orientada a objetos. Las transiciones entre las fases de análisis y diseño, y posteriormente de implementación se integran dentro de la herramienta, reduciendo apreciablemente los esfuerzos en todas las etapas del ciclo de vida del desarrollo de software. Además, incluye los últimos estándares en notación para UML [47].

Está disponible en varias ediciones, cada una destinada a unas necesidades: Enterprise, Professional, Community, Standard, Modeler y Personal. Se puede integrar con herramientas Java como: Eclipse/IBM WebSphere, JBuilder, NetBeans IDE, IntelliJ IDEA, Oracle JDeveloper y BEA Weblogic Workshop.

Existe una versión de prueba gratuita que se puede descargar desde su página web:

http://www.visual-paradigm.com/product/vpuml/

Se ha utilizado en la documentación del proyecto, con la licencia comprada por la USAL, para crear los diagramas de diseño de la interfaz en java, ya que es la única parte orientada a objetos.

4.3.8 Microsoft Proyect 2003

Proyect 2003 es una herramienta de Microsoft para realizar la planificación temporal de proyectos. Permite programar y organizar recursos y tareas, a fin de generar proyectos a tiempo y conforme a su presupuesto.

Se ha utilizado esta herramienta durante la fase de planificación del proyecto para realizar la planificación temporal ya que es una aplicación ya conocida debido a que es la que se utiliza

en las prácticas de la asignatura de segundo curso de ISI "Administración de proyectos informáticos".

4.3.9 COCOMO II

COCOMO II [49] es un modelo que permite estimar el coste, esfuerzo y tiempo cuando se planifica una nueva actividad de desarrollo software. Está asociado a los ciclos de vida modernos. El modelo original COCOMO ha tenido mucho éxito pero no puede emplearse con las prácticas de desarrollo software más recientes tan bien como con las prácticas tradicionales. COCOMO II apunta hacia los proyectos software de los 90 y de la primera década del 2000, y continuará evolucionando durante los próximos años.

La herramienta USC-COCOMO II [50] ha sido la elegida para realizar las estimaciones de coste, esfuerzo y tiempo durante la fase de planificación del proyecto. Se ha escogido esta aplicación por ser una herramienta ya conocida debido a que es la que se utiliza en las prácticas de la asignatura de segundo curso de ISI "Administración de proyectos informáticos".

4.4. Hardware y Sistema Operativo

4.4.1 En el desarrollo del software

Debido a que tanto el entorno de programación para los PICs, como el software para la programadora sólo estaban disponibles bajo Windows, se ha elaborado todo el desarrollo utilizando "Microsoft Windows XP Professional".

4.4.2 Puesta en funcionamiento del sistema

En el sistema se pueden diferenciar dos partes: la programación de los PICs y la interfaz gráfica de usuario, ejecutada en el PC.

La parte de interfaz de usuario se distribuye en forma de un único fichero *PicController.jar*. En ese archivo *jar* está contenido todo lo necesario para que la aplicación funcione. También está disponible un manual de usuario que explica las partes principales de la herramienta y sus principales características funcionales, además de cómo utilizarla correctamente.

Los únicos requisitos de software necesarios en el PC para que la interfaz funcione son:

- ✓ Tener instalado Java 2 Runtime Environment (J2RE) Standard Edition 1.5 o superior, para poder ejecutar aplicaciones Java. Se puede descargar de forma gratuita desde la página Java de Sun [12].
- ✓ Tener instalada la biblioteca RXTX (ver apartado 4.2.2), ya que es la que permite comunicarse por el puerto serie. Se puede descargar de manera gratuita en [14]. Se ha utilizado la versión para su uso sin la API Comm de Sun (espacio de nombres *gnu.io*).
- ✓ La aplicación funciona en Windows y Linux ya que tanto Java como RXTX son multiplataforma. Pero las pruebas del sistema completo sólo se han realizado en Windows debido a que ciertas partes del software de programación para PICs sólo están disponibles para esa plataforma.

Debido a que la interfaz consume pocos recursos no se han definido unos requisitos mínimos de hardware para el PC. Tan sólo es necesario que tenga disponible un puerto USB para conectar el conversor de RS485.

Para el subsistema de los PIC se necesitan:

- Microcontroladores PIC del modelo PIC18F258. Cada uno irá en una placa SBC28PC. Estarán todos conectados por el bus CAN.
- Uno de ellos realizará la adaptación entre las tramas serie y los mensajes CAN por lo que también estará conectado a un dispositivo RS485.
- Un dispositivo RS485 como por ejemplo el VScom USB-COM-I, que realiza la conversión de USB a serie, e irá conectado al ordenador por un puerto USB. Será necesario instalar el controlador para este dispositivo en el PC.
- Los sensores y actuadores, que irán conectados mediante el bus I2C al PIC que se encargue de su control. En nuestro caso basta con los SRF08.
- Los programas en ensamblador para los PICs que van a controlar los sensores o actuadores.
- El programa en ensamblador para el PIC que se encarga de realizar la conversión entre bus CAN y serie.
- Una fuente que alimente los PIC. Se recomienda alimentar el circuito con 13v.

Para mayor claridad a continuación se muestra un esquema de la arquitectura del sistema:

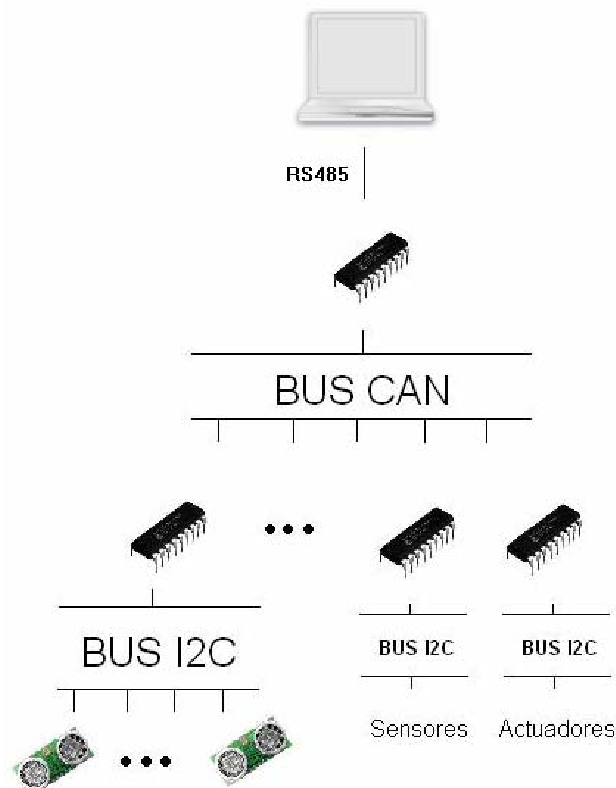


Figura 44: Esquema de la arquitectura del sistema

En la siguiente figura podemos ver un ejemplo de la interfaz cuando el sistema está en ejecución. En este caso está conectado tan sólo el nodo que controla los sónares. Este nodo solamente tiene un dispositivo SRF08 conectado con la dirección 0xE0, y está recogiendo las medidas que va tomando:

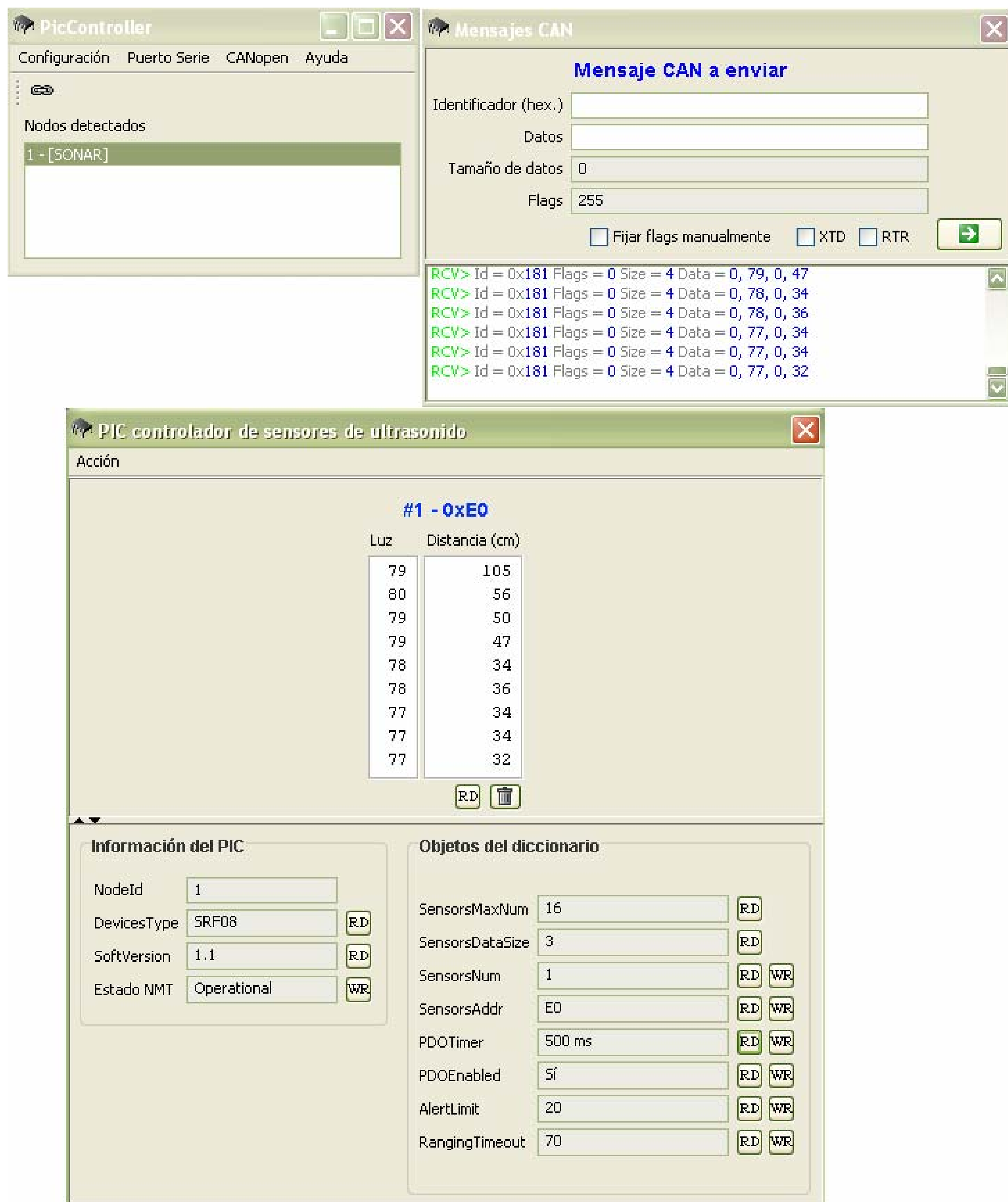


Figura 45: Ejemplo del sistema en ejecución

5. Aspectos relevantes del desarrollo

En este apartado se pretenden recoger las características más relevantes del desarrollo del proyecto, así como justificar los caminos de solución que hemos tomado para solventar los problemas más interesantes que han surgido.

Para una mejor comprensión del funcionamiento general del sistema se recomienda consultar los diagramas de actividad del “Anexo 2: Especificación de requisitos del software”, ya que en ellos se explica el recorrido de los mensajes CANopen desde que se crean en el transmisor hasta que son procesados en el receptor, y se explica más en detalle qué acciones desencadenan en función del protocolo al que correspondan.

5.1. Ciclo de vida, análisis y diseño

Se puede definir el ciclo de vida del software [5] como “las distintas fases por las que pasa el software desde que nace una necesidad de mecanizar un proceso, hasta que deja de utilizarse el software que sirvió para ese objetivo”. Este ciclo incluye, en general, una fase de requisitos, una fase de diseño, una fase de implantación, una fase de pruebas, y a veces, una fase de instalación y aceptación.

En un desarrollo iterativo e incremental, cada ciclo concluye con una versión del producto para los clientes denominada incremento. Las versiones se definen comenzando con un subsistema funcional pequeño y agregando funcionalidad con cada nueva versión. Cada ciclo consta de cuatro fases: inicio, elaboración, construcción y transición. Cada fase se subdivide a su vez en iteraciones. El resultado de una iteración es un sistema ejecutable. Cada iteración reproduce un ciclo de vida en cascada pero a una escala menor.

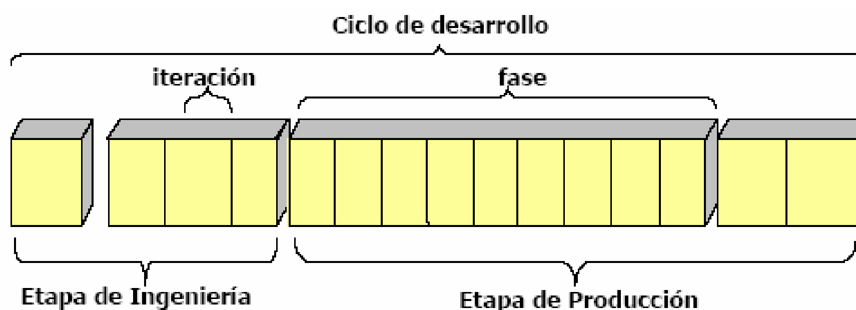


Figura 46: Ciclo de desarrollo. Proporciona un incremento.

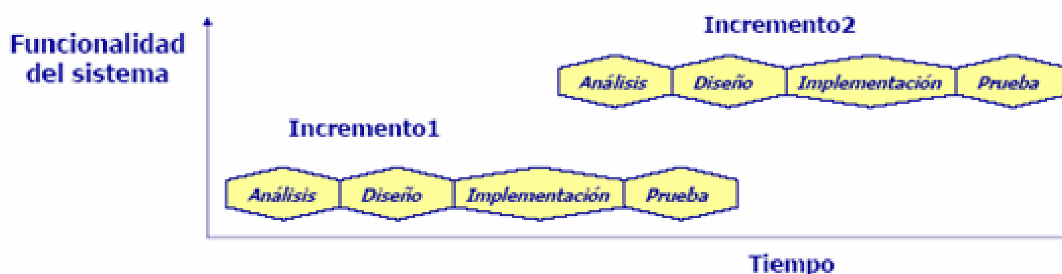


Figura 47: Proceso iterativo e incremental

Nuestro desarrollo se ajusta a un modelo de proceso iterativo e incremental, basado en la ampliación y refinamiento del sistema. En cada fase se ha realizado lo siguiente:

- Fase de inicio: hemos definido las principales funciones que debe realizar el sistema, desarrollando una descripción del producto final y una arquitectura provisional
- Fase de elaboración: se ha diseñado la arquitectura del sistema y especificado en detalle la forma de cumplir los requisitos.
- Fase de construcción: hemos creado el producto, dejándolo listo para ser utilizado. Esto no significa que estuviera completamente libre de defectos, ya que muchos de ellos se han descubierto y solucionado en la siguiente fase.
- Fase de transición: cubre el periodo durante el cual nuestro producto ha sido probado, se han corregido los problemas que ha surgido e incorporado mejoras.

En el análisis hemos puesto el énfasis en la investigación del problema y los requisitos, en vez de en la solución. En cambio durante el diseño nos hemos enfocado en la búsqueda de una solución conceptual que satisficiera los requisitos, pero no en la implementación. Se ha puesto especial atención a la definición de los objetos del sistema y en cómo colaboran para satisfacer los requisitos.

5.2. Descripción general sistema

Se ha diseñado una arquitectura en capas para que se pueda utilizar cualquier tipo de sensores y actuadores, sin tener que realizar cambios relevantes. El protocolo de comportamiento del sistema no se verá en absoluto modificado, ya que las interfaces para acceder a los servicios de cada capa están bien definidas y son independientes de los dispositivos.

Los sensores y actuadores están controlados por microcontroladores PIC, que se comunican con ellos mediante el bus I2C. A su vez los microcontroladores están comunicados entre sí mediante el bus de campo CAN.

Debido a que el bus CAN sólo define las capas física y de enlace, es necesario utilizar algún protocolo que implemente las capas superiores y especifique cómo se va a realizar la comunicación entre los distintos elementos conectados al bus. Para este fin se utiliza el

protocolo CANopen, por estar ampliamente extendido y haber sido adoptado como un estándar internacional.

CANopen se ajusta perfectamente a nuestras necesidades ya que nos proporciona un conjunto de protocolos de comunicación y servicios necesarios para implementar un sistema de control distribuido. Se pueden diferenciar dos papeles principales: el de maestro y el de esclavo. En cada sistema habrá un único maestro y uno o más esclavos.

El maestro tiene un comportamiento diferente porque es el que se encarga de toda la gestión y el único que conoce a todos los nodos que están conectados al sistema. En nuestro caso este papel lo realiza la aplicación de control que se ejecuta en el PC.

Los esclavos (representados por los PICs) no se conocen entre ellos, ya que siempre se van a comunicar con el maestro. Podemos distinguir dos grandes grupos que son los que van a marcar la principal diferencia en el comportamiento: los esclavos que controlen sensores y los que controlen actuadores. Todos los nodos esclavos van a actuar de una manera similar dependiendo de a cuál de estos dos grupos pertenezcan.

Todo el sistema se controla desde un PC, por lo que es necesario que de alguna manera éste tenga acceso al bus CAN. Para ello hay que introducir un elemento intermedio entre el bus y el ordenador, que se encargue de realizar las transformaciones necesarias. Aunque en la arquitectura final se incluirá un adaptador de USB a CAN [42] que realice estas tareas, provisionalmente se ha optado por resolver el problema de la siguiente manera. Hay un PIC adicional conectado al bus CAN cuya única finalidad es la de enviar por el puerto serie, en un formato determinado, los mensajes que le lleguen por el bus CAN, y al revés, es decir, transformar en mensajes CAN la información que le llegue por el puerto serie desde el PC. De esta forma se introduce una capa de abstracción para que el programa de control que se está ejecutando en el PC, y que actúa como maestro CANopen, pueda comportarse como un elemento más conectado al bus, cuando en realidad su comunicación se realiza por el puerto serie.

5.3. Descripción de los elementos del sistema

En el sistema hemos utilizado los siguientes buses, protocolos y dispositivos:

- Microcontroladores **PIC18F258**: Se han utilizado estos dispositivos para controlar los sensores y actuadores. Adicionalmente uno de ellos se encarga de realizar las conversiones entre bus CAN y serie, para que sea la posible la conexión con el PC. Sus características y especificaciones las encontramos en el apartado 3.1.
- Placa **SBC28PC**: Sobre este tipo de placa se montan los PICs. Sus características y especificaciones están en el apartado 3.2.
- Sensores de ultrasonidos **SRF08**: Con este tipo de sensores se han realizado todas las pruebas del sistema. Sus características y especificaciones se encuentran en el apartado 3.3.
- Bus **I2C**: Se utiliza este tipo de bus para realizar las conexiones entre los sensores y actuadores y el PIC que los controla. En el apartado 3.6 se explican en detalle sus características.
- Bus **CAN**: Todos los microcontroladores están conectados entre sí mediante este bus de campo, a una velocidad de 100Kbps. En el apartado 3.8 se explican en detalle sus características. Debido a que el ordenador no está conectado

directamente al bus CAN se utiliza un microcontrolador que actúa como mediador entre el PC y el bus, de manera que aunque el ordenador se comunica por el puerto serie, se comporta como si fuera un dispositivo más del bus.

- **CANopen**: Es el protocolo elegido para las comunicaciones entre los elementos del bus CAN. Los microcontroladores implementan los esclavos y en el PC se encuentra el maestro. En el apartado 3.10 se explican en detalle sus características.
- Conversor **VScom USB-COM-I**: Es un conversor de USB a serie. Se ha utilizado para la comunicación serie (a 19200 baudios) entre el PC y el PIC que realiza la conversión a CAN. Sus características y especificaciones están en el apartado 3.5.1.
- **PC**: En él se ejecuta la aplicación que controla todo el sistema y actúa como maestro CANopen. Se comunica con el PIC conversor mediante el puerto serie, utilizando el conversor anterior de USB a serie.

5.4. Arquitectura del sistema

Se ha diseñado una arquitectura en capas para organizar los elementos descritos anteriormente según sus responsabilidades. Esto nos permite tener cierta abstracción y más independencia. De esta manera podemos realizar cambios en las capas sin que afecten a las demás, a excepción de si se modifica la interfaz que ofrece para acceder a sus servicios.

La arquitectura se organiza en cuatro niveles:

- **Capa de control**: Es la raíz de la jerarquía y se corresponde con un PC. En él se ejecuta una interfaz gráfica desarrollada en Java para que el usuario pueda manejar y controlar todo el sistema. Esta aplicación actúa de maestro CANopen. Desde ella se puede acceder de una manera fácil e intuitiva a los servicios que este protocolo nos proporciona. Se ha diseñado de forma que cualquier usuario pueda utilizarla, aunque no tenga conocimientos sobre CANopen. La aplicación se comunica con el resto del sistema mediante mensajes CAN, pero como el PC no está directamente conectado al bus, contiene un módulo que realiza la conversión de estos mensajes a datos por el puerto serie, con un determinado formato. Por lo tanto, la comunicación con la capa inmediatamente inferior se realiza mediante la interfaz RS485.
- **Capa de adaptación de protocolos**: Está formada por un PIC que realiza la conversión entre datos serie y CAN. Se comunica con la capa de control mediante la interfaz RS485 y con la capa inmediatamente inferior por el bus CAN. Realiza las conversiones de información en ambos sentidos. Los datos serie se transmiten en un determinado formato para poder reconocer las distintas partes del mensaje CAN que contienen.
- **Capa de control de hardware**: En ella están todos los PIC del sistema que controlan los sensores y actuadores. Realizan el trabajo de un esclavo CANopen. Se comunican con los dispositivos de la capa inmediatamente inferior mediante el bus I2C, en el que el PIC actúa de maestro.
- **Capa de hardware**: Es la de más bajo nivel. En ella se encuentran los sensores y actuadores. Estos dispositivos se encargan de la interacción con el exterior. Los habrá de diversos tipos, según el propósito que deban cumplir: sónares, cámaras,

motores, etc. Todos los elementos de esta capa son esclavos de un bus I2C, por el que se comunicarán con el PIC de la capa superior que se encargue de su control.

A continuación se presenta un esquema de la arquitectura del sistema:

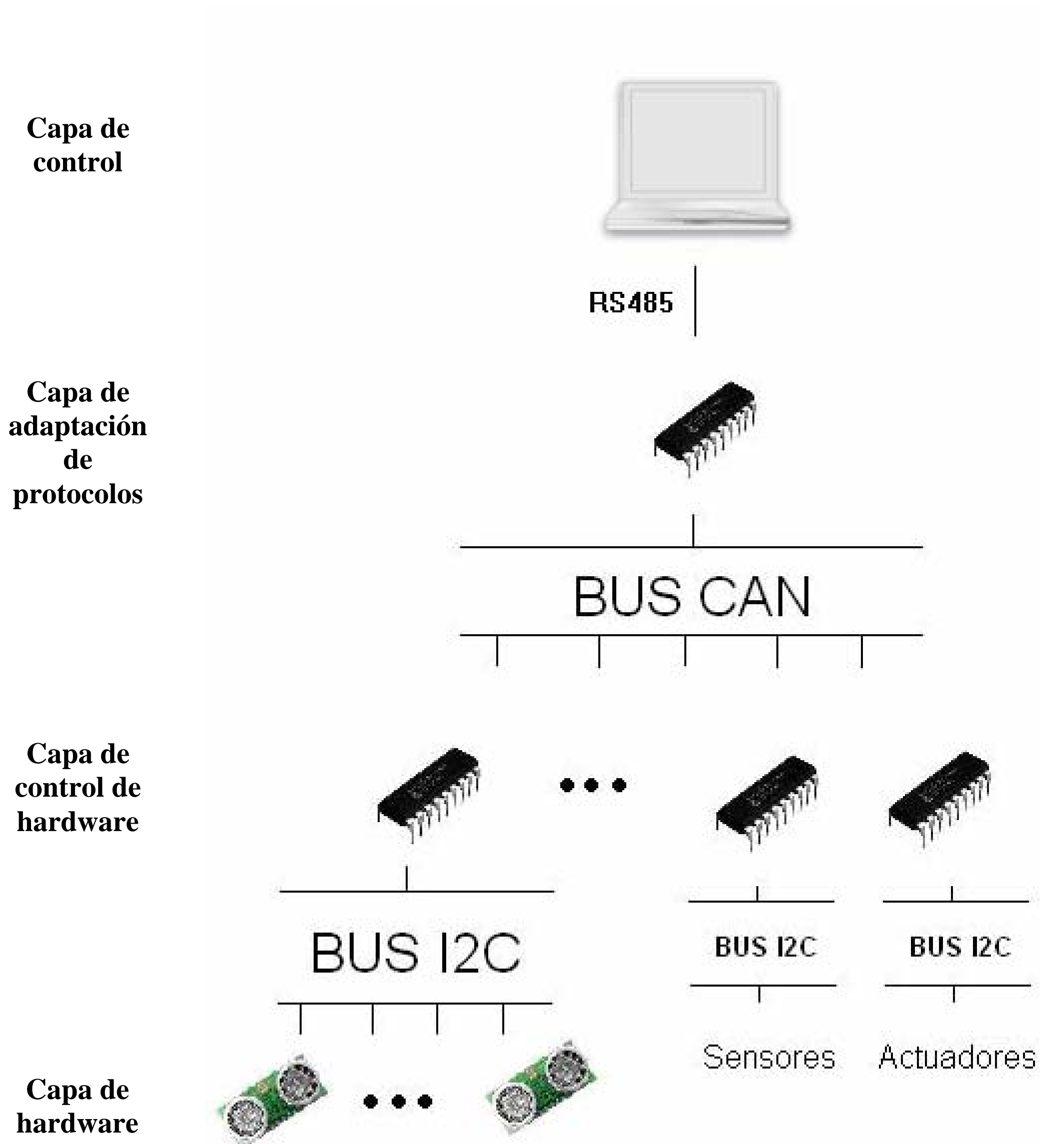


Figura 48: Esquema de la arquitectura en capas del sistema

5.5. Capas de hardware y de control de hardware

En estas capas se encuentran los sensores y actuadores y los PICs que los controlan. Cada PIC se comunica con sus dispositivos mediante el bus I2C, donde él es el maestro. Los microcontroladores están conectados entre ellos mediante el bus CAN.

Se ha diseñado e implementado de manera general una biblioteca CANopen para nodos esclavos, siendo válida para todo tipo de PICs, e independiente de los dispositivos que se conecten al nodo. Tan sólo será necesario realizar pequeños cambios, que no afectan en absoluto al protocolo ni al comportamiento general, para poder manejar características específicas de cada dispositivo. Tanto en el propio código, como en el “Anexo 4: Documentación técnica de programación”, se explica detalladamente el lugar donde se deben introducir estas pequeñas modificaciones. Se recomienda acudir a ese anexo para obtener una visión más detallada de las características de la biblioteca.

Esta biblioteca está escrita en C y se compone de los siguientes ficheros:

- **`canBus.c`**, **`canBus.h`**: Contienen la implementación que gestiona las comunicaciones por el bus CAN para los PICs de la familia PIC18FXX8. Estos ficheros deben ser modificados si se cambia de PIC. No pertenecen al protocolo CANopen, pero son necesarios para su funcionamiento.
- **`co_cancommdefs.h`**: Contiene definiciones comunes que utilizan tanto los dos ficheros siguientes, como el software que se encargue de realizar las comunicaciones por el bus CAN a bajo nivel (los dos ficheros anteriores).
- **`co_candrv.c`**, **`co_candrv.h`**: Hacen de mediador entre la biblioteca y el bus CAN. Separan la capa de más bajo nivel que se encarga de las comunicaciones por el bus y que dependerá del PIC donde se ejecute el software, de la biblioteca. Proporcionan una interfaz fija para acceder a los servicios del bus CAN de manera que se pueda cambiar tanto el PIC como el código que maneje el bus, sin necesidad de modificar nada en la biblioteca CANopen.
- **`co_dict.c`**, **`co_dict.h`**: Contienen el diccionario de objetos del nodo. El diccionario tiene partes comunes para todos los nodos, comunes para los que controlen sensores, comunes para los que controlen actuadores, y específicas del nodo. Para añadir nuevos dispositivos tan sólo es necesario modificar las específicas. Se deben incluir los objetos propios del tipo de dispositivo, así como información sobre su índice, tamaño y tipo de acceso (sólo lectura, o lectura y escritura).

  Además de contener objetos que actúan como variables, se tienen las definiciones comunes para el sistema como por ejemplo los identificadores de los mensajes CAN, entre otras cosas. Todos esos objetos son fijos y comunes para todos los nodos, incluso para el maestro.

  También contiene el identificador de nodo, que sólo es modificable directamente sobre el código fuente, por lo que debe ser cambiado antes de cargar el software en el PIC para que cada nodo tenga un identificador diferente en el sistema.
- **`co_main.c`**, **`co_main.h`**: Es el punto de acceso a la biblioteca. Contiene la función que procesa los mensajes que llegan por el bus CAN. Se encarga de revisar el tipo de mensaje que es y realizar todas las tareas correspondientes. Además tiene la función que se invoca cada vez que salta el temporizador de los PDOs, y se encarga de enviar los datos en el caso de que haya que hacerlo. Si se tuvieran más de un tipo de PDOs sería necesario añadir las funciones que se invocan al saltar sus temporizadores.

- **co_nmt.c**, **co_nmt.h**: Se encargan de toda la funcionalidad relacionada con el protocolo NMT, el cual gestiona la red y el estado de los nodos. Tan sólo contienen la parte del cliente, ya que es para los nodos esclavos.
- **co_nodeguard.c**, **co_nodeguard.h**: Se encargan de realizar el protocolo de *nodeguading*, ya que es el que se ha elegido implementar, en lugar del de *heartbeat*. Este protocolo gestiona la parte que se encarga de comprobar qué nodos están vivos y cuáles se han caído.
- **co_emergency.c**, **co_ emergency.h**: Se encargan de transmitir los mensajes de emergencia, tanto para errores ocurridos relacionados con CANopen, como para alertas de los sensores cuando sus medidas sobrepasan el límite de seguridad establecido (dato configurable, ya que es un objeto del diccionario).
- **co_sdo.c**, **co_sdo.h**: Gestionan todo lo relacionado con la recepción y el envío de SDOs. Proporcionan las primitivas para atender las peticiones del servidor y acceder al diccionario tanto para lectura como para escritura. Tan sólo contienen la parte del servidor, ya que esta biblioteca es para los nodos esclavos. Tienen implementados los cinco protocolos de petición-respuesta que se pueden aplicar a los SDOs para su transferencia.
- **co_pdo1.c**, **co_pdo1.h**: Realizan el envío de los PDO de tipo 1. Al encargarse del envío de datos recogidos por el sensor, estos ficheros deben modificarse según el tipo de dispositivos del nodo. En el caso de que algún nodo enviara más de un tipo de PDO habría que añadir los ficheros correspondientes (co_pdo2.c, co_pdo2.h, etc). El protocolo CANopen está diseñado para que puedan enviar hasta cuatro tipos de PDOs distintas.

5.5.1 Diccionario de objetos de CANopen

El diccionario de objetos es una de las partes más importantes del protocolo CANopen, ya que en él se encuentran definidos todos los objetos que se necesitan para el funcionamiento del sistema. Aunque la mayoría de objetos que contiene son comunes para todos los nodos, hay algunos específicos que deben añadirse. A pesar de que se modifiquen los objetos, las funciones de lectura y escritura en el diccionario van a permanecer invariables.

A continuación se van a ir exponiendo los objetos que forman parte del diccionario, ya sean comunes para todos los nodos, para los que controlen sensores o actuadores, o específicos del nodo.

Objetos comunes para todos los nodos

Hay ciertos objetos que están definidos para todos los nodos. Principalmente nos aportan información sobre el PIC. Estos objetos son:

- OBJ_nodeId: Identificador del nodo. Identifica a cada nodo en el sistema de una manera única. El identificador cero está reservado para el maestro.
- OBJ_softVersion: Es una cadena que contiene la versión del software del PIC.
- OBJ_devicesType: Cadena que describe el tipo de dispositivos que controla el PIC. Por ejemplo, para el PIC que controle los sónar será “SRF08”.

En la siguiente tabla encontramos información más detallada sobre ellos:

Objeto	Valor por defecto	Breve descripción	R/W	Acceso	Tamaño (en bytes)	Índice
OBJ_nodeId	Ej. “1”	Identificador del nodo	RO	SDO	1	1200
OBJ_softVersion	Ej. “1.1”	Versión del software del PIC	RO	SDO	3	1201
OBJ_devicesType	Ej. “SRF08”	Tipo de dispositivos que controla	RO	SDO	5	1202

Tabla 13: Objetos del diccionario comunes para todos los nodos

El valor por defecto (y por lo tanto el tamaño) de estos objetos varía según el nodo, y debe ser modificado directamente en el código fuente, ya que durante la ejecución permanecerá invariable.

Objetos comunes para los nodos que controlen sensores

Cada nodo tendrá conectado un grupo de dispositivos que se encargará de una determinada tarea. Los que tengan conectados sensores van a actuar de una manera muy similar. El esquema general de funcionamiento es:

- En el bucle principal del programa se van a dedicar a leer continuamente los datos recogidos por los sensores.
- Tendrán un temporizador (Timer0) que cada cierto tiempo provocará una interrupción y en ese momento se realizará el envío de datos, si se cumplen las condiciones.
- Las condiciones para el envío de datos son:
  - Que el nodo se encuentre en el estado *Operational*. Esta condición también debe cumplirse cuando el envío se hace como resultado de la llegada de una trama RTR.
  - Que el envío de datos a través de PDOs (cuando salta el temporizador) esté permitido. Lo está siempre que el *flag* `pdoEnabled` esté activo.
- El valor del temporizador se dará en milisegundos y por defecto es de medio segundo.
- Cada vez que se reciba un mensaje por el bus CAN se provocará una interrupción. El PIC recogerá el mensaje y dejará que la biblioteca de CANopen lo procese.

Las definiciones comunes para los sensores son:

- `OBJ_sensorsMaxNum`: Indica el máximo número de sensores conectados al PIC. Al conectarse mediante el bus I2C este número va a ser 16, ya que es el máximo número que permite este tipo de bus.
- `OBJ_sensorsNum`: Indica el número de sensores conectados actualmente.
- `OBJ_sensorsAddr`: Es una matriz que contiene las direcciones de los sensores conectados. El orden en el que se introducen las direcciones es importante ya que

va a ser el orden de lectura. Por ejemplo en el caso de los sónares se toman mediciones de dos en dos, por lo que es recomendable que se introduzcan pares de sensores enfrentados para evitar interferencias.

- `OBJ_pdoTimer`: Contiene el valor del temporizador en milisegundos. Indica cada cuánto tiempo se va a producir el envío de los PDOs.
- `OBJ_pdoEnabled`: *Flag* que indica si se deben enviar o no los PDOs cada vez que salte el temporizador. Si está activado se deben enviar.
- `OBJ_sensorsDataSize`: Indica el número de *bytes* de datos que recoge un sensor.

En la siguiente tabla encontramos información más detallada sobre ellos:

Objeto	**Valor por defecto**	**Breve descripción**	**R/W**	**Acceso**	**Tamaño (en bytes)**	**Índice**
OBJ_sensorsMaxNum	16	Número máx. de sensores	RO	SDO	1	1203
OBJ_sensorsNum	1	Número de sensores	RW	SDO	1	1204
OBJ_sensorsAddr [ ]		Direcciones de los sensores	RW	SDO	OBJ_sensorsMaxNum	1205
OBJ_pdoTimer	0,5s	Temporizador para PDOs	RW	SDO	2	1206
OBJ_pdoEnabled	1	Permitido envío de PDOs	RW	SDO	1	1207
OBJ_sensorsDataSize	Ej. 3	Bytes de datos leídos por los sensores	RO	SDO	1	1208

Tabla 14: Objetos del diccionario comunes para los nodos que controlan sensores

Objetos comunes para los nodos que controlen actuadores

Los que tengan conectados actuadores también van a actuar de una manera muy similar. Se dedicarán a escuchar los mensajes que le lleguen por el bus CAN y procesarlos mediante la librería de CANopen. Su principal tarea es esperar los datos que le lleguen mediante PDOs y realizar con los actuadores las acciones pertinentes.

Las definiciones comunes para los actuadores son:

- `OBJ_actsMaxNum`: Indica el máximo número de actuadores conectados al PIC. Al conectarse mediante el bus I2C este número va a ser 16, ya que es el máximo número que permite este tipo de bus.
- `OBJ_actsNum`: Indica el número de actuadores conectados actualmente.
- `OBJ_actsAddr`: Es una matriz que contiene las direcciones de los actuadores conectados.

- `OBJ_actsDataSize`: Indica el número de *bytes* de datos que espera cada actuador.

En la siguiente tabla encontramos información más detallada sobre ellos:

Objeto	**Valor por defecto**	**Breve descripción**	**R/W**	**Acceso**	**Tamaño (en bytes)**	**Índice**
OBJ_actsMaxNum	16	Número máx. de actuadores	RO	SDO	1	1220
OBJ_actsNum	1	Número de actuadores	RW	SDO	1	1221
OBJ_actsAddr [ ]		Direcciones de los actuadores	RW	SDO	OBJ_actsMaxNum	1222
OBJ_actsDataSize	Ej. 2	Bytes de datos para los actuadores	RO	SDO	1	1223

Tabla 15: Objetos del diccionario comunes para los nodos que controlan actuadores

Objetos específicos de los nodos que controlen sensores de ultrasonido

El sensor de ultrasonido utilizado en el sistema es el SRF08. Este sensor tarda en tomar las medidas entre 65-70ms desde de que se lanza a medir. Para optimizar los tiempos de medición se ha decidido lanzar a medir los sensores en pares de dos enfrentados. De esta manera se optimiza el tiempo de recogida de datos de todos los sónares.

Por esta razón es necesario definir en el diccionario el siguiente objeto:

- `OBJ_rangingTimeout`: Determina el tiempo en milisegundos que tarda un sensor en tomar una medida. Es decir, el tiempo que pasa desde que se comienza la medición hasta que tiene los datos.

En la siguiente tabla encontramos información más detallada:

Objeto	**Valor por defecto**	**Breve descripción**	**R/W**	**Acceso**	**Tamaño (en bytes)**	**Índice**
OBJ_rangingTimeout	70	Tiempo de toma de medidas del sónar	RW	SDO	1	1220

Tabla 16: Objetos del diccionario para los nodos que controlen sensores de ultrasonido

5.6. Capa de adaptación de protocolos

Esta capa contiene el PIC que se encarga de realizar las conversiones entre datos serie y mensajes del bus CAN. Toda la información necesaria para poder reconstruir el mensaje CAN original se encapsula dentro de una trama que se transmite por el puerto serie. Durante la etapa de "diseño de datos" se determinó cuál debería de ser el formato de estas tramas:

- Delimitador de inicio (4 *bytes*): DLE (0x10), SYN (0x16), DLE, SYN.
- Número de *buffer* de recepción (1 *byte*): Puede tener tres valores diferentes. Si es "0" ó "1" indica en qué *buffer* del PIC se recibió el mensaje. Si es "2" significa que es un mensaje CAN a transmitir.
- Identificador del mensaje CAN (4 *bytes*): Debido a que los mensajes CAN pueden tener identificadores de hasta 29 bits, se necesitan 4 *bytes* para transmitir esta información. Para ello se utiliza el sistema *big-endian*, es decir, primero el *byte* más significativo o de mayor peso.
- *Flags* (1 *byte*): Este campo contiene los *flags* del mensaje CAN.
- Tamaño de los datos (1 *byte*): Indica el tamaño de los datos del mensaje CAN. Como en el bus CAN sólo se pueden transmitir hasta 8 *bytes* de datos, es suficiente con un *byte* para este campo.
- Datos (hasta 8 *bytes*): Son los *bytes* de datos del mensaje CAN.
- Delimitador de fin (2 *bytes*): DLE, ETX (0x03).
- CRC (1 *byte*): Campo que se utiliza para comprobar que la trama se ha transmitido correctamente y la información no se ha corrompido o es errónea. El CRC (código de redundancia cíclica) se obtiene realizando la operación XOR (OR exclusivo) de todos los *bytes* de la trama.

Para detectar correctamente los delimitadores de inicio y fin, incluso aunque su secuencia esté repetida en los datos de la trama, se agrega un DLE por cada DLE que aparezca en los datos. Esta tarea se lleva a cabo justo antes de transmitir la trama, una vez que ya ha sido generada. También se realiza en sentido inverso, eliminando uno de los DLEs que forman parte de los datos y que se reciben duplicados. De esta forma la trama entregada al nivel superior es la misma que la original. Al ser una labor que pertenece a la capa de enlace, los DLEs duplicados no se contabilizan en el cálculo del CRC, ya que no forman parte de la trama en sí.

Tanto en la capa de control como en esta de adaptación de protocolos, existe un autómata finito que se encarga de ir detectando las tramas completas que van llegando, y extrayendo los mensajes CAN que contienen. En el caso de que el CRC recibido no coincidiera con el calculado a partir de los datos que han llegado, la trama completa se deshecha, ya que se considera que no es correcta.

5.7. Capa de control

En esta capa se encuentra la aplicación de control. Es una interfaz gráfica de usuario desarrollada en Java. Con ella se puede controlar y monitorizar todo el sistema. Implementa a un maestro CANopen. Actúa como si estuviera conectada directamente al bus CAN gracias a que se ha introducido una capa de abstracción que se encarga de realizar las conversiones entre datos.

5.7.1 Descripción de la aplicación de control

La aplicación de control ha sido diseñada siguiendo el patrón Modelo-Vista-Controlador [3]. De esta manera se separa la lógica de negocio de la forma de representar los datos, mediante los controladores. La aplicación posee dos controladores importantes: uno para mediar con la biblioteca de CANopen y otro para el puerto serie. De esta forma se abstrae a la interfaz de las particularidades de ambos protocolos de comunicación, permitiendo hacer cambios en cualquiera de las partes sin afectar a las demás.

Podemos dividirla en tres módulos:

- **Puerto serie**: Se encarga de la comunicación por el puerto serie. También realiza la conversión entre los datos que se transmiten por él y los mensajes CAN que representan. Para ello existe una clase que posee un autómata finito que reconoce el formato de los datos del puerto serie y extrae de ellos los mensajes CAN cuando están completos. En ese momento avisa a la interfaz y a la biblioteca de CANopen de la llegada del mensaje. También se encarga de realizar la transformación en el sentido inverso.

  La comunicación entre el resto de la aplicación y la clase que se encarga de las comunicaciones por el puerto serie siempre se realiza a través del controlador del puerto serie. Al introducir esta capa de abstracción se logra un desacoplo entre las partes.

- **Biblioteca CANopen**: Biblioteca que implementa a un maestro CANopen. Los mensajes CAN que lleguen se envían a su clase principal, la cual se encarga de reconocer a qué protocolo pertenece y entregárselo al módulo que le corresponda manejarlo. Los eventos ocurridos se los comunica a la interfaz la realiza mediante el controlador de CANopen. De esta manera se logra el desacoplo entre ambas. Los mensajes CAN los envía mediante el controlador del puerto serie. La funcionalidad de esta biblioteca se explica con más detalle en el apartado 5.7.3.

- **Interfaz gráfica**: Se encarga de la representación de la información. Los controladores le notifican los eventos que van ocurriendo y ella se encarga de realizar las actualizaciones necesarias en los elementos de representación. También mediante estos controladores notifica a los otros dos módulos los eventos relacionados con ellos que el usuario provoca. Gracias a la introducción de esta capa de abstracción se logra el desacoplo entre las distintas partes.

5.7.2 Breve descripción de la interfaz

A continuación, se van a presentar brevemente las principales partes que forman la interfaz y sus funcionalidades. Al arrancar la aplicación nos encontraremos lo siguiente:

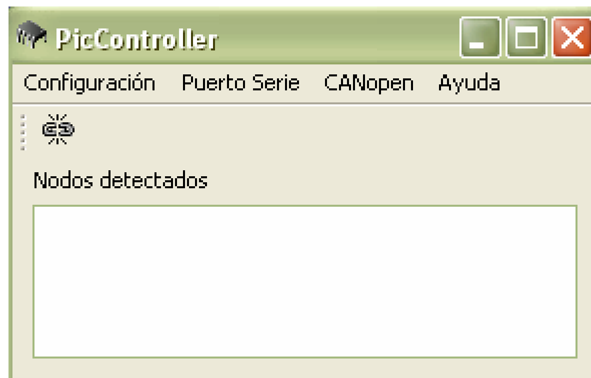


Figura 49: Aspecto de la aplicación al arrancar

En la barra de herramientas encontramos un botón que nos servirá para conectar y desconectar del puerto serie. Una vez conectados, en la lista de nodos detectados irán apareciendo los PICs que se vayan conectando al sistema, es decir, los diferentes PICs de la capa de control de hardware.

A continuación se explicará la funcionalidad de los menús más importantes.

Puerto Serie

En este menú encontramos tres opciones:

- Una para configurar los parámetros del puerto serie. Ello se realiza mediante el cuadro de diálogo de la siguiente figura. En parámetro "Puerto" sólo aparecerán los puertos que se encuentren disponibles en el sistema en ese momento.

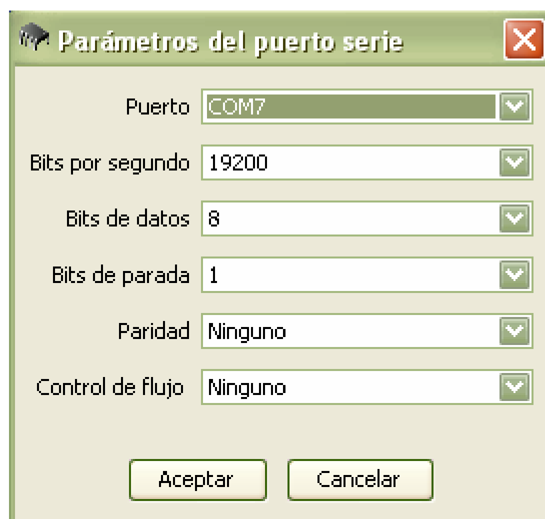


Figura 50: Configuración del puerto serie en Windows

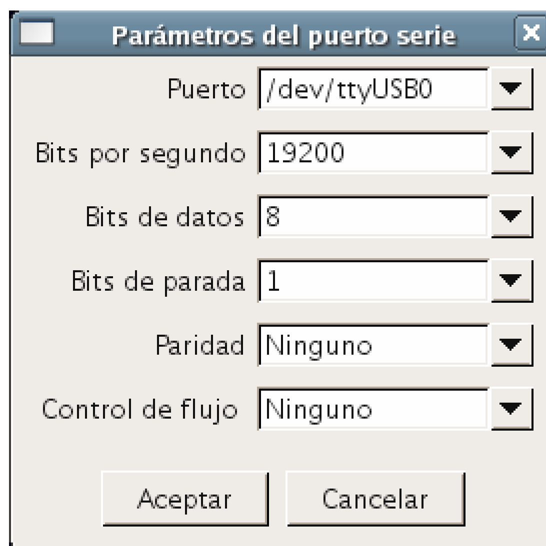


Figura 51: Configuración del puerto serie en Linux

- También desde allí se puede acceder la ventana que muestra toda la comunicación por el puerto serie, tal y como está ocurriendo. Desde él también se pueden enviar *bytes* de información en decimal. Si estos datos se introdujeran de forma incorrecta (como muestra el ejemplo "Figura 52") se resaltarían en rojo para avisar al usuario.

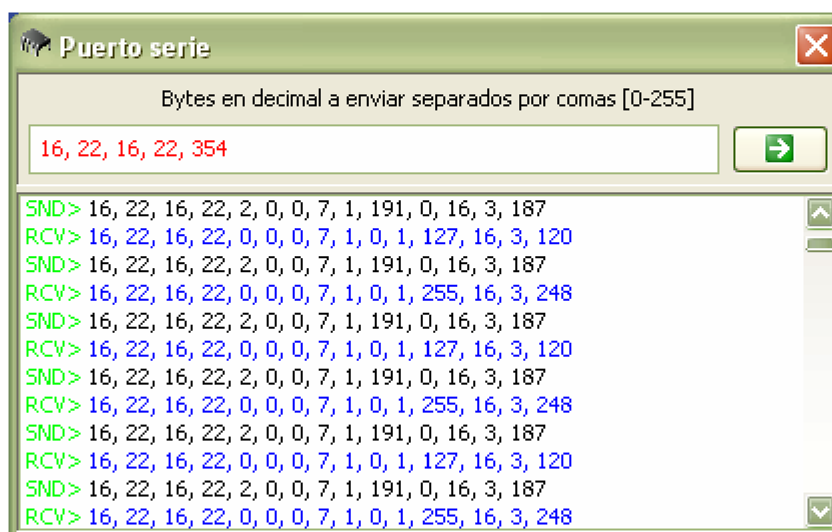


Figura 52: Comunicación por el puerto serie

- Por último accedemos la ventana que muestra los mensajes CAN que se están enviando y recibiendo. Es decir, como si la interfaz formara directamente parte del bus CAN. La herramienta ayuda al usuario desglosando cada mensaje en sus partes fundamentales y mostrando información sobre sus *flags* de forma textual (indica cada *flag* que esté activado). También permite enviar mensajes CAN. Un mensaje CAN tiene tres partes fundamentales:
  - El identificador: Se va a introducir en hexadecimal para un reconocimiento más sencillo del protocolo CANopen al que pertenece.
  - Los datos: Hasta ocho *bytes* en decimal. Si se introdujeran erróneamente la aplicación los resaltaría en rojo.
  - El tamaño de los datos: La interfaz lo deduce automáticamente según el usuario va introduciendo los datos.
  - Los *flags*: Se pueden introducir de manera automática mediante casillas de verificación, o de forma manual.

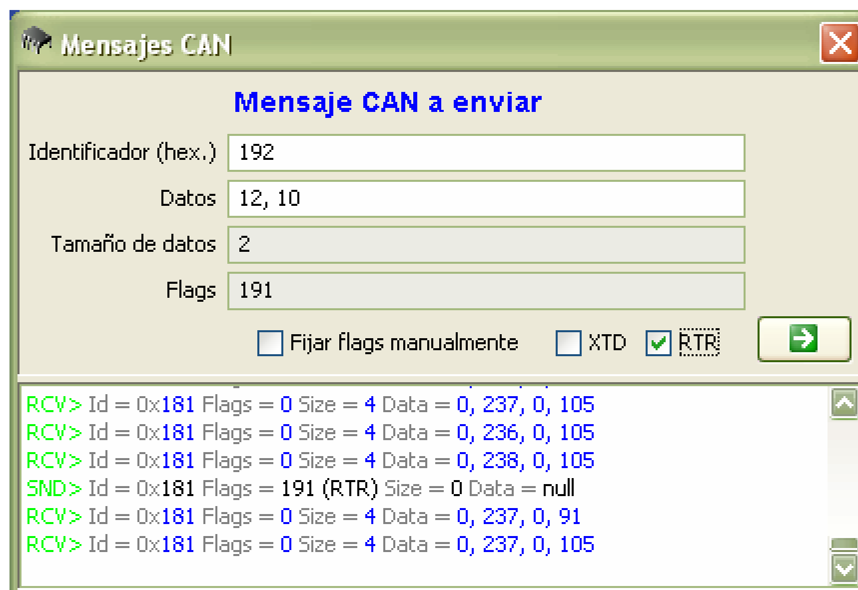


Figura 53: Mensajes CAN enviados y recibidos

CANopen

Bajo este menú encontramos tres apartados:

- Uno para configurar los parámetros del diccionario de objetos de cada nodo del sistema. Esto se hace a través del cuadro de diálogo de la figura. En el cuadro combinado aparecen los nodos que están conectados actualmente en el sistema. La zona inferior va cambiando dinámicamente según se van seleccionando los nodos, de manera que muestra los objetos configurables que cada uno tiene en su diccionario. Toda esta información es almacenada en un fichero *confFileNodeX.properties*, donde X es el número de nodo (ver apartado 5.7.5).

Figura 54: Editor de los ficheros de configuración de los nodos

- También desde ese menú se puede acceder a los mensajes de emergencia de CANopen que se han recibido. Para ello hay una ventana que se activa cada vez que llega un nuevo mensaje (contiene una opción para desactivar este comportamiento). En ella se muestra la información del mensaje de una manera comprensible para el usuario, aunque no tenga conocimientos sobre CANopen. El controlador de eventos de CANopen es el que se encarga de extraer la información relevante del mensaje, codificada según especifica el formato de los mensajes de emergencia. Adicionalmente se puede mostrar también el mensaje CAN tal cual llega, para un usuario más avanzado.

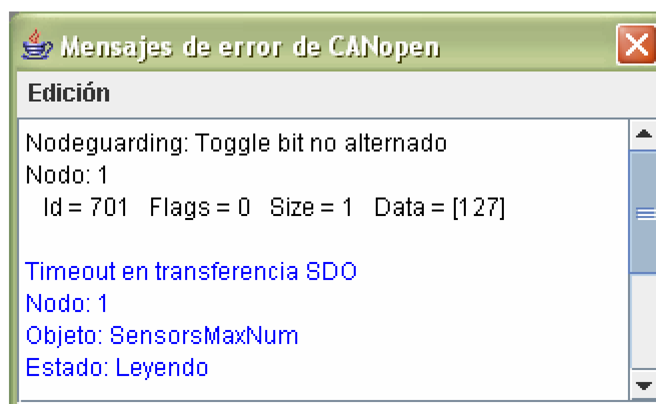


Figura 55: Mensajes de error de CANopen

- Por último también permite acceder a otro tipo de mensajes de emergencia, las alertas. Las envían los PICs que controlan sensores cuando sus mediciones no se encuentran dentro de los límites de seguridad establecidos. Por ejemplo el nodo de los sónares las transmite al acercarse demasiado a un objeto. También en esta ventana se muestra la información de un modo fácilmente comprensible por el usuario. El comportamiento es el mismo que en el caso anterior, ya que ambos son mensajes de emergencia.

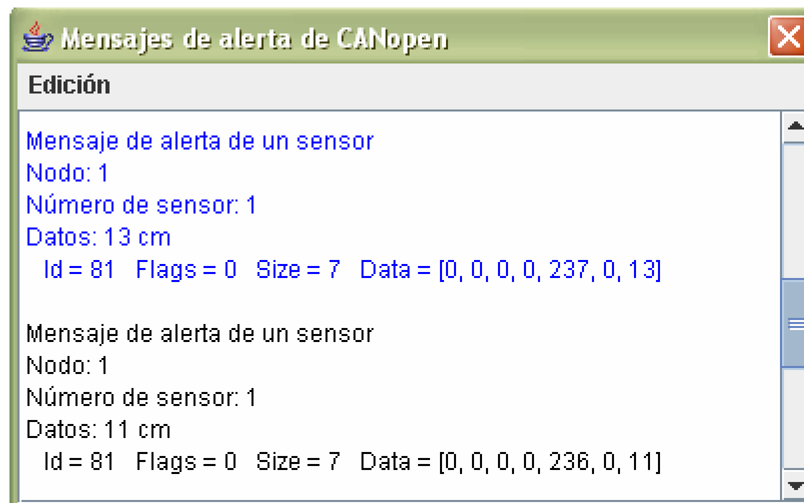


Figura 56: Mensajes de alerta de CANopen

PICs

Cuando un PIC se conecta al sistema aparece en la lista de nodos detectados de la ventana principal. Se muestra información sobre el número de nodo y el tipo de dispositivos que controla:

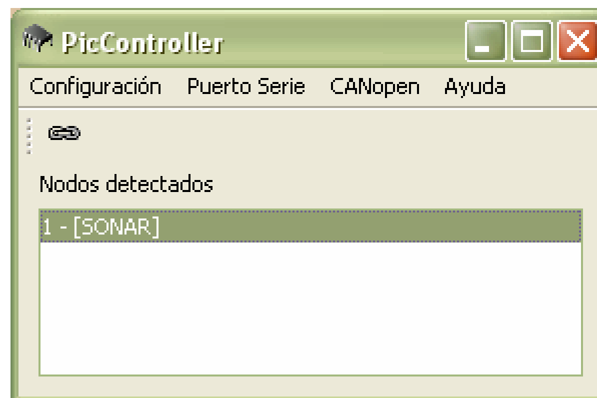


Figura 57: Ventana principal con un nodo conectado al sistema

Si se hace doble clic sobre el nodo aparecerá una ventana, específica para cada tipo de dispositivo controlado, con los objetos de su diccionario:

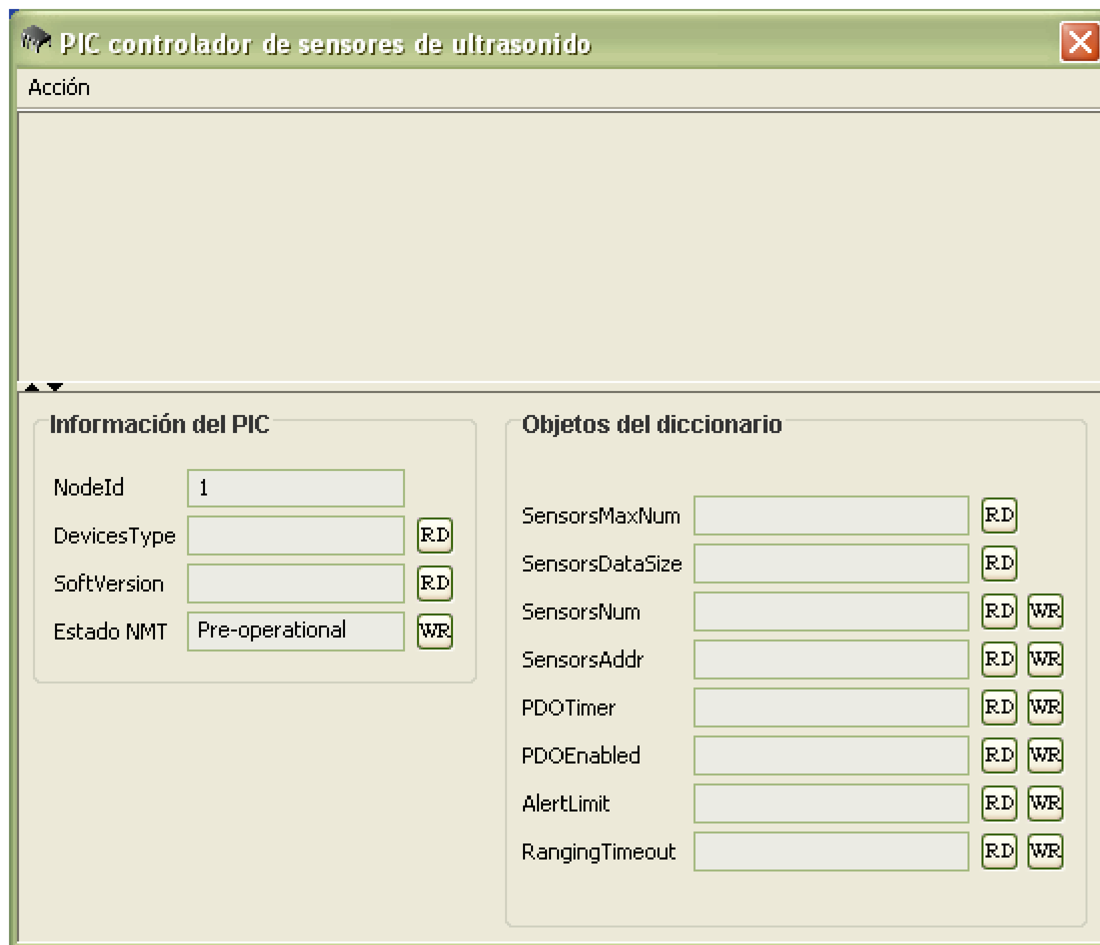


Figura 58: Controlador para sónares antes de configurar el PIC

A través de esa ventana se pueden leer y configurar los objetos del diccionario del PIC. En este caso controla sónares y como mínimo es necesario introducir la dirección de los sensores que tenga conectados para que comience a tomar medidas. Esto se puede hacer manualmente o de forma automática cargando el fichero de configuración del nodo.

Para mayor rapidez y comodidad existe un menú “Acción” con las siguientes opciones:

- **Cargar el fichero de configuración**: Configura el nodo. Para todos los objetos configurables del diccionario del PIC escribe los valores almacenados en el fichero de configuración propio del nodo. Esto lo realiza mediante mensajes SDO.
- **Leer todos los parámetros**: Envía todos los mensajes SDO necesarios para leer cada objeto del diccionario del PIC.
- **Enviar trama RTR**: Envía un mensaje PDO de retransmisión para que el PIC transmita los últimos valores leídos por los sensores.

También se puede realizar la lectura o escritura individual de los objetos del diccionario del PIC mediante los botones “**RD**” y “**WR**” asociados a cada parámetro. Los que sean directamente configurables por el usuario se pueden modificar mediante cuadros de diálogo diseñados para editar un número, un valor *booleano* o direcciones de dispositivos. Todos están

preparados para que el usuario sólo pueda introducir valores correctos. Si se salen de rango o son erróneos por otras razones, se le avisará mediante diferentes mecanismos. A continuación tenemos varios ejemplos de estos tipos de diálogos:

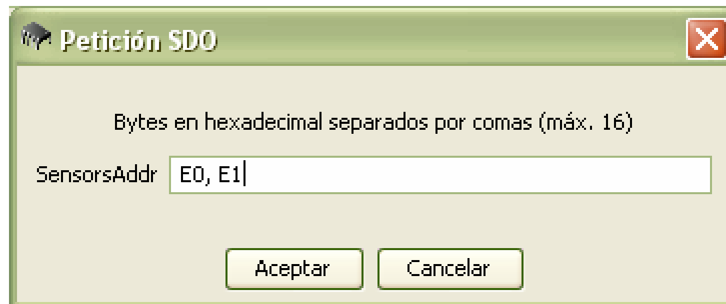


Figura 59: Configurar la dirección de los sensores

Figura 60: Configurar el objeto *PdoEnabled*

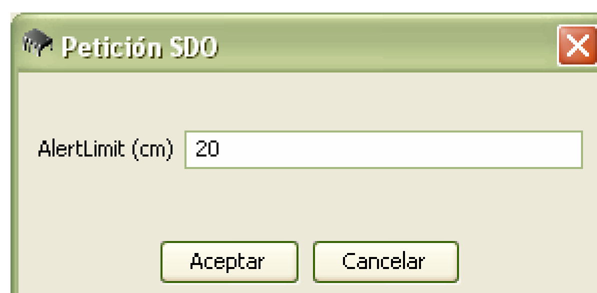


Figura 61: Configurar el objeto *AlertLimit*

Una vez configurado el PIC, éste se puede arrancar, ordenándole pasar al estado "Operational" ("Figura 63"). En ese momento comenzará a transmitir las medidas que sus sensores estén tomando. En el ejemplo "Figura 62" se puede observar un controlador para sensores SRF08 con un sólo dispositivo conectado con dirección 0xE0. El PIC ya está configurado y en estado "Operational", por lo que se están recibiendo las medidas tomadas por el sónar.

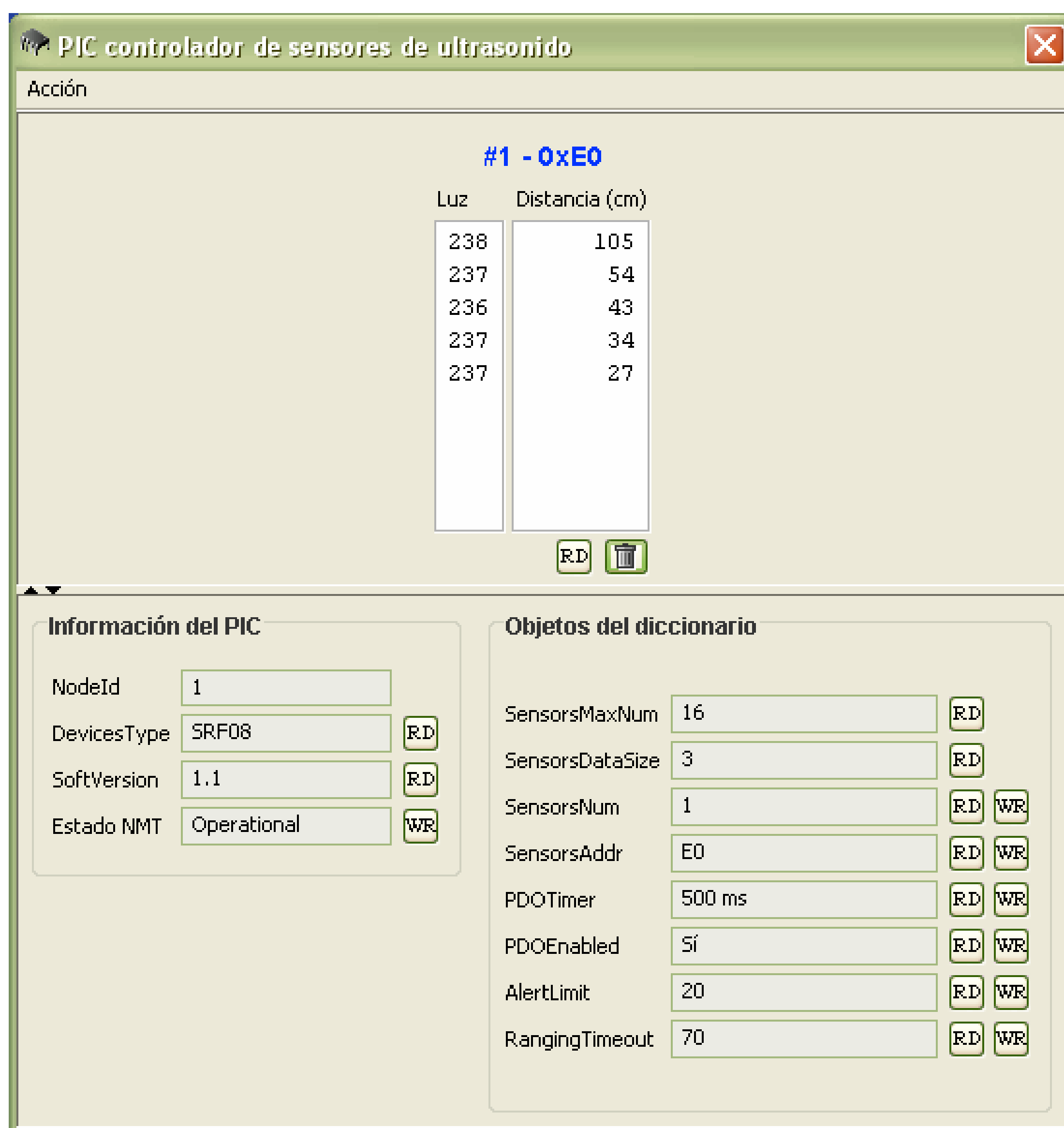


Figura 62: Ejemplo del controlador para sónares recibiendo medidas

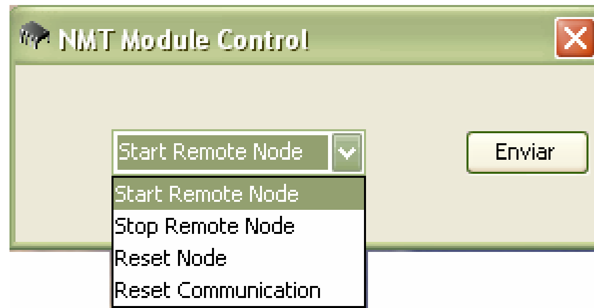


Figura 63: Cambio del estado NMT

En la figura anterior se muestra el cuadro de diálogo que permite cambiar el estado NMT del PIC. Tan sólo estarán disponibles las opciones que se puedan aplicar en cada momento.

5.7.3 Biblioteca CANopen

Una parte fundamental de la herramienta es la que contiene la biblioteca de CANopen, ya que es la que se encarga de manejar todos los eventos relacionados con este protocolo. Existe una clase para gestionar cada tipo de mensaje: NMT, SDO, PDO, etc.

Para una visión más en profundidad de la biblioteca se recomienda acudir al “Anexo 4: Documentación técnica de programación”. A continuación se van a explicar brevemente algunas de las partes principales:

- **Clase principal de CANopen**: Es la clase que se encarga de procesar cada mensaje CAN que llega. Decide a qué protocolo pertenece y a qué clase le delega su procesamiento. Es la única clase que se comunica con el exterior de la biblioteca. Se encarga de crear un controlador de CANopen para cada nodo conectado al sistema, del tipo que sea necesario. La labor de estos controladores se explica más adelante. No se deben confundir con los cuadros de diálogo de la interfaz, los cuales tan sólo se dedican a mostrar información y gestionar eventos del usuario.
- **Módulo NMT**: Se encarga de realizar el protocolo NMT, el cual gestiona la red y el estado de los nodos. Manda los mensajes correspondientes para cambiar el estado NMT de los PICs.
- **Módulo de *nodeguarding***: Se encarga de realizar el protocolo de *nodeguading*, ya que es el que se ha elegido implementar, en lugar del de *heartbeat*. Este protocolo gestiona la parte que se encarga de comprobar qué nodos están vivos (y cuál es su estado NMT) y cuáles se han caído. Para ello envía cada cierto tiempo un mensaje de *nodeguarding* al nodo. Si éste lo contesta es que está vivo. Si pasado un cierto tiempo no ha recibido respuesta, lo considera caído. En ambos casos avisa a la interfaz de que actualice el estado del nodo mediante el controlador de eventos para CANopen. Para gestionar los tiempos de espera, tanto entre peticiones como para la respuesta, se utilizan hilos. Mientras se reciban mensajes del nodo se interpretará que éste está vivo, y no se emitirán este tipo de mensajes.
- **Módulo para SDOs**: Gestiona todo lo relacionado con la transferencia y envío de SDOs. Implementa la parte del cliente, que es el que realiza las peticiones de lectura o escritura en el diccionario de cada PIC. Tienen implementados los cinco protocolos de petición-respuesta que se pueden aplicar a los SDOs para su transferencia.

Para evitar que la interfaz se quede colapsada durante el tiempo que duran estas transferencias, este módulo se implementa en forma de hilo. Contiene una cola en la cual se van añadiendo y sacando peticiones, y otra donde se almacenan los mensajes SDO que van llegando. Se utilizan mecanismos de sincronización para el control de accesos concurrentes, y un semáforo para poder realizar espera bloqueante sin consumir ciclos de CPU.

También se utilizan hilos para los tiempos de espera de la transferencia. En el caso de que no se obtenga respuesta después de un cierto tiempo, se informará a la interfaz de ello a través del controlador de eventos para CANopen.

- **Módulo para PDOs**: Se encarga de gestionar las recepciones de los PDOs y los envíos de tramas de retransmisión para solicitar a los PICs la transferencia de PDOs. Debido a que la interpretación de los datos que contienen es diferente para cada tipo de nodo, existe una interfaz que deben implementar las clases que gestionen las PDOs de cada tipo de PIC.
- **Módulo de mensajes de emergencia**: Se encarga de recibir los mensajes de emergencia, tanto para errores ocurridos relacionados con CANopen, como para alertas de los sensores cuando sus medidas sobrepasan el límite de seguridad establecido (dato configurable, ya que es un objeto del diccionario del PIC). Estos mensajes son entregados al controlador de eventos de CANopen, quien los interpreta y entrega a la interfaz en forma de mensaje comprensible por cualquier usuario. También se muestra el mensaje recibido tal cual para usuarios avanzados.
- **Módulo de comunicaciones por el bus CAN**: Se encarga de enviar los mensajes CAN. Hace de mediador entre la biblioteca CANopen y el controlador del puerto serie, a quien le pasa los mensajes CAN a transmitir. De esta forma aunque hubiera cambios en el controlador del puerto serie, la biblioteca CANopen no se vería afectada.
- **Diccionario de objetos**: Contiene un paquete con todos los tipos de objetos que puede haber en los diccionarios de los nodos. También tiene una clase con todas las definiciones generales de sistema, y que forman parte del diccionario del nodo maestro. Casi todas ellas también deben aparecer en los diccionarios de los nodos esclavos.
- **Controladores CANopen para cada tipo de PIC**: Adicionalmente existen controladores para cada tipo de nodo que puede haber en el sistema, y que dependen de la naturaleza de los dispositivos que controle: si son sensores o actuadores. Existe una interfaz que todos deben implementar para que aunque la forma de acceder a los servicios que ofrecen sea común, la manera de atenderlos sea diferente y específica de cada dispositivo. Además de éstos métodos necesarios para realizar el control del PIC, también contienen su diccionario de objetos, con los parámetros propios de cada nodo.

5.7.4 Internacionalización

La internacionalización [15] es el proceso de diseñar una aplicación para que pueda ser adaptada a diferentes idiomas y regiones, sin necesitad de cambios de ingeniería. Un programa internacionalizado tiene las siguientes características:

- Los elementos textuales como mensajes de estado y etiquetas de elementos de la interfaz no están codificadas dentro del programa. Son almacenados fuera del código fuente y recuperados de forma dinámica.
- El soporte de nuevos idiomas no requiere re-compilación.
- Otros datos dependientes de la cultura, como fechas y monedas, aparecen en el formato e idioma de la región del usuario final.
- Puede ser localizado rápidamente. La localización es el proceso de adaptar software para una región o idioma específico añadiendo componentes específicos de la localidad y traduciendo el texto. Como ya hemos dicho, todo el texto de la interfaz se almacena fuera del código fuente y es recuperado en tiempo de ejecución. Por esta razón, el programa no requiere ninguna modificación. Los traductores trabajan con ficheros de texto que son leídos por la aplicación pero no están dentro de ella.
- Con la adición de datos de localización, el mismo ejecutable funciona en cualquier lugar del mundo.
- El texto mostrado por el programa está en el idioma nativo del usuario final.

Se ha implementado la interfaz de manera que está disponible tanto en inglés como en español, y además, es fácilmente ampliable a otros idiomas. La aplicación ha sido internacionalizada por lo que el adaptarla y portarla para otros idiomas resulta muy sencillo. Además, el idioma de la interfaz se puede cambiar en el acto y sin tener que reiniciarla.

Esto se ha conseguido con la siguiente técnica. El texto de la interfaz en un idioma se almacena en un fichero de propiedades, con formato de texto plano. En el nombre del fichero va incluido el código del idioma al que pertenece: se le añaden un par de letras minúsculas conformes a la norma ISO-639. Hay una lista completa de códigos ISO-639 en *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*. Nosotros tenemos dos archivos: *AppText_en.properties* y *AppText _es.properties*, para inglés y español. Su estructura es muy simple. Están formados por parejas clave-valor. Las claves tienen que ser las mismas en ambos ficheros ya que es a lo que se hace referencia desde el programa para recuperar el texto. A continuación podemos ver una pequeña parte estos archivos:

AppText_en.properties	**AppText_es.properties**
SerialPort=Serial port	SerialPort=Puerto serie
Edit=Edit	Edit=Edici\u00F3n
OkButton=Ok	OkButton=Aceptar

Accedemos al texto desde dentro del código de la aplicación de la siguiente manera:

1. Recuperamos el idioma que ha elegido el usuario del fichero de configuración (esto se explicará más adelante en el apartado 5.7.5):

```
String language = Preferences.getValue("language");
```

2. Creamos el objeto `ResourceBundle`. Estos objetos se utilizan para aislar los datos sensibles a la localidad, como texto traducible, etc. En nuestro programa, el `ResourceBundle` está constituido por los ficheros de propiedades que contienen los mensajes que queremos mostrar.

   ```
   appText = ResourceBundle.getBundle(
                "picController/resources/AppText",
                new Locale(language));
   ```

3. Ahora cada vez que queramos recuperar un texto del archivo lo hacemos utilizando el objeto `ResourceBundle` *`appText`*. Por ejemplo, para recuperar la cadena del mensaje de error "No se puede enviar el mensaje" haríamos:

   ```
   appText.getString("ERCantSendMsg")
   ```

De esta manera, cuando el usuario decida cambiar el idioma, tan sólo habría que volver a cargar los textos de todos los elementos de la interfaz, lo que es instantáneo y no supone ningún problema.

Si se quisiera ampliar el número de idiomas de la interfaz, tan sólo habría que crear un nuevo fichero de propiedades traducido al lenguaje correspondiente, lo cual es fácil y rápido.

A continuación podemos ver algunos ejemplos de la interfaz al cambiar el idioma:

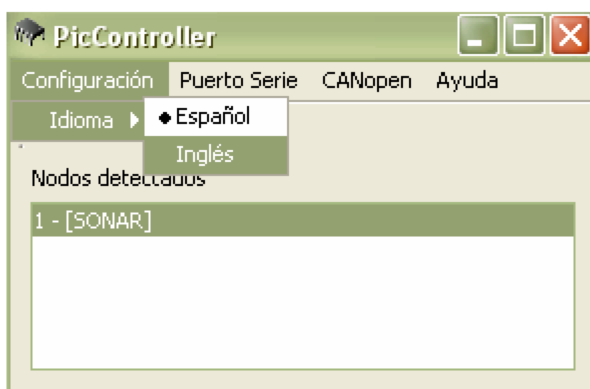


Figura 64: Cambio de idioma

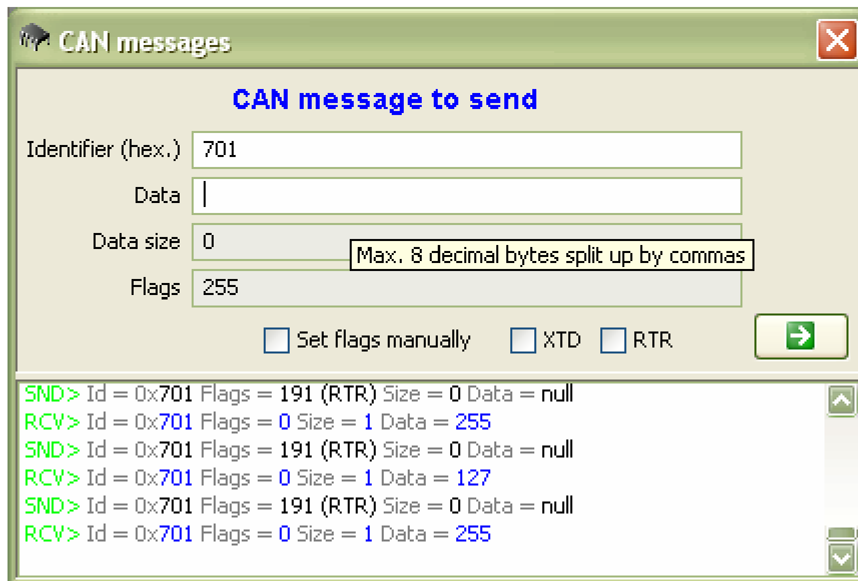


Figura 65: Cuadro de diálogo para mensajes CAN en inglés

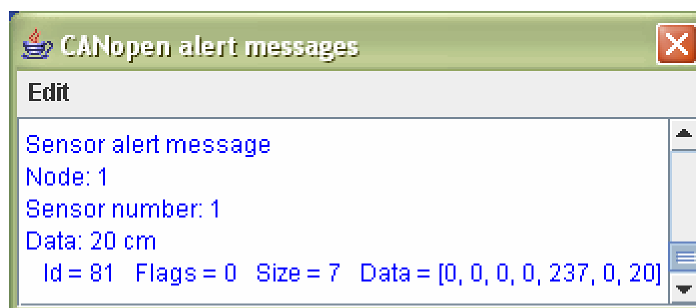


Figura 66: Mensajes de alerta en inglés

5.7.5 Configuración de la interfaz

Hay varias opciones configurables en la interfaz como son: el idioma, todos los parámetros del puerto serie, y los valores de los objetos del diccionario de CANopen para cada nodo. Los valores establecidos por el usario se conservan entre las diferentes ejecuciones del programa. Los dos primeros se almacenan en un fichero llamado *configPicController.properties*. Los valores de los objetos del diccionario para cada nodo se almacenan en un fichero llamado *configFileNodeX.properties*, donde *X* es el número del nodo. De esta manera tenemos un fichero diferente para cada nodo.

Estos archivos se crean en el directorio *PicControllerConfigFiles*, el cual se genera dentro del directorio donde está el ejecutable. Cuando se ejecuta el programa por primera vez, o no se encuentra el archivo, se crea un fichero de configuración con unas opciones por defecto, evitando así errores.

Estos archivos tienen la misma estructura que los ficheros de idiomas explicados anteriormente. Están formados por parejas clave-valor. Se han creado una clase `Preferences` y otra `COConfigFiles`, las cuales se encargan de acceder a ellos, recuperando o modificando los valores para las claves, y generando nuevos archivos con valores por defecto si no se encuentran. La primera accede al fichero de configuración de la interfaz *configPicController.properties*, y la segunda a cada fichero de configuración de cada nodo. Hacen de enlace entre la aplicación y los archivos.

Por ejemplo, para recuperar o almacenar el valor de una opción configurable, la clase `Preferences` sigue estos pasos:

1. Carga el archivo de configuración. Si no lo encuentra genera uno nuevo por defecto dentro del directorio *PicControllerConfigFiles*. Si tampoco encuentra esta carpeta, también la genera, dentro del directorio actual en el cual se está ejecutando la aplicación. Almacena los valores en el objeto *properties*, de tipo `Properties`.

   ```
   FileInputStream fis = new FileInputStream(file);
   properties.load(fis);
   fis.close();
   ```

2. Si se quiere recuperar el valor de una propiedad, cuya clave es *key*, se hace:

   ```
   String value = (String) properties.getProperty(key);
   ```

   En cambio, si lo que se pretende es almacenar un valor, se haría lo siguiente:

   ```
   properties.setProperty(key, value);
   properties.store(
     new FileOutputStream(new File(CONF_FILE)), null);
   ```

   Siendo *key* la clave de la opción que se quiere almacenar, *value*, su valor, y *CONF_FILE* el nombre del fichero de configuración (incluyendo el directorio dentro del que se encuentra, de manera relativa al directorio actual).

La clase `COConfigFiles` seguiría unos pasos parecidos, pero tendría en cuenta el tipo de objetos que tiene cada nodo, según su número de identificación, el cual forma parte del nombre del archivo. Cada vez que se quiere acceder o modificar una propiedad, además de pasar como parámetro la clave, se pasa el número de nodo. De esta manera la clase puede `COConfigFiles` conoce tanto el nombre del fichero al que acceder como los objetos configurables de cada nodo.

6. Trabajos relacionados

Se están desarrollando de forma paralela varios proyectos relacionados con otros elementos de la carretilla. Uno de los mayores problemas de los robots móviles es el de la localización dentro de un espacio de trabajo. En relación con este tema existen varios trabajos diferentes:

- En uno de ellos se trabaja con cámaras web que irán situadas encima de la carretilla, y que consiste en saber en qué posición se encuentra mediante el reconocimiento de patrones. Estos patrones se distribuyen por el espacio de trabajo, conociendo de antemano su posición y orientación. Se utiliza la biblioteca ARToolKit [52] para ayudar a detectarlos, y que nos devuelva su posición y orientación relativas con respecto al robot. De esta manera se puede deducir la localización en el espacio de trabajo.
- En otro se trabaja con el *Pan and Tilt* sobre se colocarán las cámaras web. Se está desarrollando una biblioteca de funciones para gobernar su movimiento, de manera que se le pueda indicar un ángulo y una dirección y el dispositivo se mueva.
- También se están desarrollando trabajos para hacer la cámara web siga un patrón. Es decir, una vez que lo detecta, sigue su movimiento.
- Existe otro trabajo que utiliza el láser que irá en la carretilla. Se pretende emplear este sensor para determinar qué puntos forman una línea recta, y así reconocer paredes y esquinas. Mediante técnicas de SLAM (*Simultaneous Localization and Mapping*) se realizan labores de posicionamiento y construcción de mapas de entrono a partir de las medidas proporcionadas por el láser.

Además de estos proyectos, previamente se realizó un estudio sobre los sónares SRF08 que se utilizarán en la carretilla, y con los que se han realizado las pruebas de este proyecto. Este estudio formó parte de un trabajo de doctorado en el que se pretendía modelar estos sensores. Conocer cómo funcionan, sus características, el rango de sus mediciones y los factores que les influyen.

7. Conclusiones

Tras finalizar este proyecto hemos comprobado que se han cumplido satisfactoriamente todos y cada uno de los objetivos que teníamos. Se ha desarrollado una arquitectura software y hardware para la recogida de los datos sensoriales y el control de los actuadores.

Durante el desarrollo se han ido adquiriendo los conocimientos y experiencias necesarios para crear cada parte del sistema, dando como resultado una arquitectura en capas flexible y bien definida, así como un software robusto y de calidad.

Como protocolo de nivel superior que especifique la manera de realizar la comunicación entre los distintos elementos conectados al bus se ha elegido e implementado CANopen, ya que está ampliamente extendido y ha sido adoptado como un estándar internacional. Debido a que la documentación que existía estaba en inglés y no resultaba muy clara, se ha realizado una en castellano que será utilizada en desarrollos posteriores.

Se ha desarrollado una biblioteca de funciones que implementa en C un esclavo CANopen. Esta biblioteca puede reutilizarse en cuantos esclavos CANopen sean necesarios, independientemente del tipo de dispositivos que controlen. Tan sólo habrá que hacer pequeñas modificaciones para añadir características específicas de cada modelo de sensor o actuador. Cómo y dónde realizar estos cambios está explicado con detalle tanto en el código fuente directamente, como en la documentación de la API entregada en formato digital, y en el anexo de "Documentación técnica de programación" de este documento.

Adicionalmente se ha implementado en Java el maestro CANopen, el cual se encarga del control del sistema. Desde la interfaz gráfica el usuario puede monitorizar e interaccionar con el sistema. Puede comprobar qué nodos están conectados al bus CAN, qué tipo de dispositivos controlan, cuál es su estado, si están funcionando correctamente tanto ellos como sus dispositivos, puede enviar órdenes y modificar parámetros del diccionario de objetos.

La interfaz abstrae al usuario de las peculiaridades del bus CAN y CANopen, siendo sencilla e intuitiva, pudiendo ser utilizada por cualquier persona que no conozca los protocolos. Pero a su vez permite a los usuarios con un cierto grado de conocimientos saber qué es lo que está pasando en el sistema en cada momento, ya que toda la comunicación tanto por el bus CAN como por el puerto serie se encuentra monitorizada, y existe la posibilidad de enviar directamente mensajes CAN, u otros datos por el puerto serie.

También se ha implementado la capa intermediaria entre los dispositivos del bus CAN, y el PC por el puerto serie. De esta manera el sistema se comporta como si la aplicación de control que está en el PC y realiza las labores de maestro CANopen fuera un nodo más del bus CAN.

Se ha realizado una documentación completa tanto de los conceptos teóricos relacionados con el desarrollo, como de las bibliotecas de funciones y programas implementados, para su reutilización o ampliación en el futuro.

En lo que se refiere al ámbito personal también se han cumplido mis objetivos, ya que he sido capaz de asimilar en poco tiempo muchos conceptos, y llevarlos a la práctica con éxito. Además he logrado superar las dificultades que entrañan la programación a tan bajo nivel, la depuración de los programas de los PICs, y el comprender e implementar un protocolo tan amplio y peculiar como CANopen a partir de la documentación existente. Finalmente ha sido una experiencia muy gratificante y que me ha aportado muchos nuevos conocimientos.

8. Líneas de trabajo futuras

En el futuro se pretenden añadir nuevos nodos al sistema que controlen otros tipos de dispositivos, por lo que será necesario generar los nuevos programas que harán de esclavos de CANopen e irán incluidos en cada PIC. Para ello se utilizará la biblioteca de CANopen ya creada.

También será necesario añadirlos en la aplicación en Java. Además se quiere incluir una imagen de la carretilla en la que se pueda especificar dónde está situado cada sensor y que represente de forma gráfica los datos o medidas que esté tomando.

9. Bibliografía y referencias

[1] **García, Francisco J., Maudes, Jesús M., Piattini, Mario G., García-Bermejo, José R. y Moreno, María N.** "*Proyecto de Final de Carrera en la Ingeniería Técnica en Informática: Guía de Realización y Documentación*". Versión 1.52, 18 de Marzo de 2000.

[2] **García, Francisco J., González, Guillermo, Moreno, María de N., Moreno, Vidal.** "*Normas para la Elaboración de Informes Técnicos en el Departamento de Informática y Automática de la Universidad de Salamanca*".

[3] **Larman, Craig.** "*UML y Patrones. Una introducción al análisis y diseño orientados a objetos y al Proceso Unificado*". Segunda edición. Prentice Hall. 2003.

[4] **Jacobson, Ivar, Booch, Grady, Rumbaugh, James.** "*El Lenguaje Unificado de Modelado. Manual de Referencia*". Addison Wesley. 2000.

[5] **García, Francisco J.** *Apuntes de la asignatura "Ingeniería del software"*. 3º de I.T.I.S. Curso 2003-2004. Universidad de Salamanca.

[6] **García, Francisco J.** *Apuntes de la asignatura "Programación orientada a objetos"*. 3º de I.T.I.S. Curso 2003-2004. Universidad de Salamanca.

[7] **Deitel, H. M., Deitel, P. J.** "*Cómo programar en Java*". Prentice Hall, Primera edición, 1998.

[8] **Jaworski, Jamie.** "*Java 1.2 Al descubierto*". Prentice Hall, 1999.

[9] **Eckel, Bruce.** "*Piensa en Java*". Prentice Hall. Segunda edición, 2002.

[10] **Lemay, Laura, Cadenhead, Rogers.** "*Java 2 in 21 Days*". SAMS Teach Yourself. SAMS, 2002.

[11] **Lemay, Laura, Cadenhead, Rogers.** "*Java 2 in 24 Hours*". SAMS Teach Yourself. SAMS, 2002.

[12] **Sitio Java de Sun.**
http://java.sun.com/ (Última vez visitado: 27/6/2007)

[13] **Java Communications API.**
http://java.sun.com/products/javacomm/ (Última vez visitado: 27/6/2007)

[14] **RXTX.**
http://rxtx.org/ (Última vez visitado: 2/8/2005)

[15] **Java en Castellano.**
http://www.programacion.com/java/tutoriales/ (Última vez visitado: 27/6/2007)

[16] **Entorno de desarrollo Eclipse.**
http://www.eclipse.org/ (Última vez visitado: 27/6/2007)

[17] **Doxygen.**
http://www.doxygen.org/ (Última vez visitado: 27/6/2007)

[18] **Página web de SourceBoost.**
http://www.sourceboost.com/ (Última vez visitado: 27/6/2007)

[19] **Entorno de desarrollo SourceBoost IDE.**
http://www.sourceboost.com/Products/SourceBoostIde/Overview.html
(Última vez visitado: 27/6/2007)

[20] **Compilador BoostC.**
http://www.sourceboost.com/Products/BoostC/Overview.html
(Última vez visitado: 27/6/2007)

[21] **Página web de Microchip.**
http:// www.microchip.com/ (Última vez visitado: 27/6/2007)

[22] **Página web de Intel.**
http://www.intel.com/ (Última vez visitado: 27/6/2007)

[23] **Página web de Freescale.**
http://www.freescale.com/ (Última vez visitado: 27/6/2007)

[24] **Software para PICkit 2.**
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en023805 (Última vez visitado: 27/6/2007)

[25] **Programadora PICkit 2.**
http://www.modtronix.com/product_info.php?products_id=257
(Última vez visitado: 27/6/2007)

[26] **Adaptador PGM2KIT.**
http://www.modtronix.com/product_info.php?products_id=258
(Última vez visitado: 27/6/2007)

[27] **PIC18F258.**
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1335&dDocName=en010281 (Última vez visitado: 27/6/2007)

[28] **PIC18FXX8 Data Sheet.**
http://ww1.microchip.com/downloads/en/DeviceDoc/41159e.pdf
(Última vez visitado: 27/6/2007)

[29] **Especificaciones del Bus I2C.** "The I2C bus specification". Semiconductores Philips. Versión 2.1. *http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf*
(Última vez visitado: 27/6/2007)

[30] **CiA (CAN in Automation).**
http://www.can-cia.org/ (Última vez visitado: 27/6/2007)

[31] **Bus CAN.**
http://www.can-cia.org/can/standardization/standards.html/
(Última vez visitado: 27/6/2007)

[32] **Protocolo CANopen.** "*CANopen: high level protocol for CAN-bus*".
http://www.nikhef.nl/pub/departments/ct/po/doc/CANopen30.pdf
(Última vez visitado: 27/6/2007)

[33] **CANopen solutions.**
http://www. canopensolutions.com/index.html (Última vez visitado: 27/6/2007)

[34] **Placa SBC28PC.**
http://www.modtronix.com/product_info.php?products_id=111
(Última vez visitado: 27/6/2007)

[35] **Sensor SRF08.**
http://www.superrobotica.com/S320112.htm
(Última vez visitado: 27/6/2007)

[36] **Bus.**
http://es.wikipedia.org/wiki/Bus_%28Inform%C3%A1tica%29
(Última vez visitado: 27/6/2007)

[37] **Bus de campo.**
http://en.wikipedia.org/wiki/Fieldbus
http://es.wikipedia.org/wiki/Bus_de_campo
(Última vez visitados: 27/6/2007)

[38] **Bus I2C.**
http://robots-argentina.com.ar/Comunicacion_busI2C.htm
(Última vez visitado: 27/6/2007)

[39] **RS485.**
http://es.wikipedia.org/wiki/RS-485 (Última vez visitado: 27/6/2007)

[40] **VScom USB-COM-I.**
http://www.vscom.de/produkte/vscom_usb-com-i.html (Última vez visitado: 27/6/2007)

[41] **232Analyzer.**
http://www.232analyzer.com/RS232_Protocol_Analyzer_Monitor/RS232_Analyzer_Monitor_DOWNLOAD.HTM (Última vez visitado: 27/6/2007)

[42] **Adaptador de USB a CAN.**
http://www.canusb.com/ (Última vez visitado: 27/6/2007)

[43] **Microcontroladores.**
http://es.wikipedia.org/wiki/Microcontrolador (Última vez visitado: 27/6/2007)

[44] **DSP (Procesador digital de señal).**
http://es.wikipedia.org/wiki/Procesador_digital_de_se%C3%B1al
(Última vez visitado: 27/6/2007)

[45] **Página web de Molex.**
http://www.molex.com/ (Última vez visitado: 27/6/2007)

[46] **Lenguaje de programación C.**
http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_C
(Última vez visitado: 27/6/2007)

[47] **Visual Paradigm for UML.**
http://www.visual-paradigm.com/product/vpuml/ (Última vez visitado: 27/6/2007)

[48] **UML.**
http://www.uml.org/ (Última vez visitado: 27/6/2007)

[49] **Moreno, Ana M.** "*Estimación de proyectos software*".

[50] **COCOMO II.**
http://sunset.usc.edu/research/COCOMOII/cocomo_main.html
(Última vez visitado: 27/6/2007)

[51] **Microsoft Proyect 2003.**
http://www.microsoft.com/spain/office/products/project/default.mspx/
(Última vez visitado: 27/6/2007)

[52] **ARToolKit.**
http://www.hitl.washington.edu/artoolkit (Última vez visitado: 27/6/2007)